Amazon Athena: User Guide
Copyright © 2022 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.
# Table of Contents

What is Amazon Athena? ........................................................................................................... 1
When should I use Athena? ......................................................................................................... 1
  Amazon Athena ....................................................................................................................... 1
  Amazon EMR .......................................................................................................................... 2
  Amazon Redshift .................................................................................................................... 2
Accessing Athena ...................................................................................................................... 2
Understanding tables, databases, and the Data Catalog .......................................................... 3
AWS service integrations with Athena ....................................................................................... 4
**Setting up** ............................................................................................................................... 7
  1. Sign up for an AWS account ............................................................................................ 7
  2. Create an IAM administrator user and group ................................................................. 7
  3. Attach managed policies for Athena ............................................................................... 7
  4. Sign in as an IAM user ..................................................................................................... 8
**Getting started** ..................................................................................................................... 9
  Prerequisites .......................................................................................................................... 9
  Step 1: Create a database ..................................................................................................... 9
  Step 2: Create a table .......................................................................................................... 12
  Step 3: Query data .............................................................................................................. 16
  Saving your queries ............................................................................................................. 18
  Connecting to other data sources ....................................................................................... 18
**Accessing Amazon Athena** .................................................................................................. 19
  Using the console ............................................................................................................... 19
  Using the API ....................................................................................................................... 19
  Using the CLI ....................................................................................................................... 20
**Connecting to data sources** ................................................................................................ 21
  Integration with AWS Glue .................................................................................................. 21
    Using AWS Glue to connect to data sources in Amazon S3 .............................................. 22
    Registering an AWS Glue Data Catalog from another account ......................................... 24
    Best practices when using Athena with AWS Glue .......................................................... 25
    Upgrading to the AWS Glue Data Catalog step-by-step ................................................... 31
    FAQ: Upgrading to the AWS Glue Data Catalog ............................................................... 33
  Using a Hive metastore ......................................................................................................... 35
    Overview of features ......................................................................................................... 35
    Workflow ............................................................................................................................ 36
    Considerations and limitations ......................................................................................... 37
    Connecting Athena to an Apache Hive metastore ............................................................. 38
    Using the AWS Serverless Application Repository .......................................................... 41
    Connecting Athena to Hive using an existing role ............................................................ 43
    Configure Athena to use a deployed Hive metastore connector ..................................... 46
    Using a default data source ............................................................................................... 47
    Hive views .......................................................................................................................... 50
    Using the AWS CLI with Hive metastores ....................................................................... 54
    Reference implementation ................................................................................................. 59
**Using Amazon Athena Federated Query** ........................................................................... 60
  Considerations and limitations ............................................................................................. 61
  Videos .................................................................................................................................. 62
  Deploying a connector and connecting to a data source ....................................................... 62
  Using the AWS Serverless Application Repository ............................................................. 64
  Creating a VPC for a data source connector ....................................................................... 66
  Athena data source connectors ............................................................................................ 67
  Writing federated queries .................................................................................................... 167
  Enabling cross-account federated queries .......................................................................... 171
  Writing a data source connector ......................................................................................... 177
**IAM policies for accessing data catalogs** .......................................................................... 178
Compression support in Athena by file format .......................................................... 300
Compression support in Athena engine version 3 .................................................. 300
Compression support in Athena engine version 2 .................................................. 300
Specifying compression formats ........................................................................... 301
Specifying no compression .................................................................................... 301
Notes and resources ............................................................................................... 301
SerDe reference ...................................................................................................... 302
Using a SerDe ......................................................................................................... 302
To use a SerDe in queries ....................................................................................... 302
Supported SerDes and data formats ...................................................................... 303
Amazon Ion Hive SerDe ......................................................................................... 305
Avro SerDe ............................................................................................................. 315
Grok SerDe ............................................................................................................. 317
JSON SerDe libraries ............................................................................................. 319
LazySimpleSerDe for CSV ................................................................................... 324
OpenCSVSerDe ...................................................................................................... 329
ORC SerDe ............................................................................................................. 332
Parquet SerDe ........................................................................................................ 334
RegEx SerDe ......................................................................................................... 337
Running queries .................................................................................................... 339
Viewing query plans .............................................................................................. 340
See Also ................................................................................................................ 344
Query results and recent queries ........................................................................... 344
Specifying a query result location ......................................................................... 344
Downloading query results files using the Athena console .................................... 347
Viewing recent queries ......................................................................................... 349
Finding query output files in Amazon S3 .............................................................. 352
Reusing query results ............................................................................................ 354
Key features .......................................................................................................... 354
Considerations and limitations ............................................................................. 355
Reusing query results in the Athena console ......................................................... 355
Viewing query stats .............................................................................................. 357
See also ................................................................................................................ 361
Working with views ............................................................................................... 361
When to use views? ............................................................................................... 362
Supported actions for views in Athena .................................................................. 362
Considerations for views ....................................................................................... 363
Limitations for views ............................................................................................ 364
Working with views in the console ...................................................................... 364
Creating views ....................................................................................................... 365
Examples of views ................................................................................................. 367
Using saved queries .............................................................................................. 368
Considerations and limitations ............................................................................. 368
Working with saved queries in the Athena console .............................................. 369
Using the Athena API to update saved queries .................................................... 370
Using parameterized queries .............................................................................. 370
Considerations and limitations ............................................................................ 370
Querying using execution parameters .................................................................. 371
Querying with prepared statements .................................................................... 372
See also ................................................................................................................ 376
Handling schema updates ..................................................................................... 376
Summary: Updates and data formats in Athena .................................................... 377
Index access in ORC and parquet ......................................................................... 378
Types of updates .................................................................................................... 379
Updates in tables with partitions .......................................................................... 384
Querying arrays ...................................................................................................... 385
Creating arrays ...................................................................................................... 385
<table>
<thead>
<tr>
<th>Chapter Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenating arrays</td>
<td>387</td>
</tr>
<tr>
<td>Converting array data types</td>
<td>388</td>
</tr>
<tr>
<td>Finding lengths</td>
<td>388</td>
</tr>
<tr>
<td>Accessing array elements</td>
<td>389</td>
</tr>
<tr>
<td>Flattening nested arrays</td>
<td>390</td>
</tr>
<tr>
<td>Creating arrays from subqueries</td>
<td>392</td>
</tr>
<tr>
<td>Filtering arrays</td>
<td>393</td>
</tr>
<tr>
<td>Sorting arrays</td>
<td>394</td>
</tr>
<tr>
<td>Using aggregation functions with arrays</td>
<td>394</td>
</tr>
<tr>
<td>Converting arrays to strings</td>
<td>395</td>
</tr>
<tr>
<td>Using arrays to create maps</td>
<td>396</td>
</tr>
<tr>
<td>Querying arrays with complex types and nested structures</td>
<td>397</td>
</tr>
<tr>
<td>Querying geospatial data</td>
<td>402</td>
</tr>
<tr>
<td>What is a geospatial query?</td>
<td>402</td>
</tr>
<tr>
<td>Input data formats and geometry data types</td>
<td>403</td>
</tr>
<tr>
<td>Supported geospatial functions</td>
<td>403</td>
</tr>
<tr>
<td>Examples: Geospatial queries</td>
<td>418</td>
</tr>
<tr>
<td>Querying Hudi datasets</td>
<td>420</td>
</tr>
<tr>
<td>Hudi dataset table types</td>
<td>420</td>
</tr>
<tr>
<td>Considerations and limitations</td>
<td>422</td>
</tr>
<tr>
<td>Video</td>
<td>422</td>
</tr>
<tr>
<td>Creating Hudi tables</td>
<td>422</td>
</tr>
<tr>
<td>See also</td>
<td>425</td>
</tr>
<tr>
<td>Querying JSON</td>
<td>425</td>
</tr>
<tr>
<td>Best practices for reading JSON data</td>
<td>426</td>
</tr>
<tr>
<td>Extracting data from JSON</td>
<td>427</td>
</tr>
<tr>
<td>Searching for values in JSON arrays</td>
<td>429</td>
</tr>
<tr>
<td>Obtaining length and size of JSON arrays</td>
<td>431</td>
</tr>
<tr>
<td>Troubleshooting JSON queries</td>
<td>432</td>
</tr>
<tr>
<td>Using ML with Athena</td>
<td>432</td>
</tr>
<tr>
<td>Considerations and limitations</td>
<td>432</td>
</tr>
<tr>
<td>ML with Athena syntax</td>
<td>433</td>
</tr>
<tr>
<td>Customer use examples</td>
<td>434</td>
</tr>
<tr>
<td>Querying with UDFs</td>
<td>434</td>
</tr>
<tr>
<td>Considerations and limitations</td>
<td>434</td>
</tr>
<tr>
<td>Videos</td>
<td>435</td>
</tr>
<tr>
<td>UDF query syntax</td>
<td>435</td>
</tr>
<tr>
<td>Examples</td>
<td>437</td>
</tr>
<tr>
<td>Creating and deploying a UDF using Lambda</td>
<td>437</td>
</tr>
<tr>
<td>Querying across regions</td>
<td>442</td>
</tr>
<tr>
<td>Considerations and limitations</td>
<td>442</td>
</tr>
<tr>
<td>Querying AWS Glue Data Catalog</td>
<td>442</td>
</tr>
<tr>
<td>Considerations and limitations</td>
<td>442</td>
</tr>
<tr>
<td>Listing databases and searching a specified database</td>
<td>443</td>
</tr>
<tr>
<td>Listing tables in a specified database and searching for a table by name</td>
<td>444</td>
</tr>
<tr>
<td>Listing partitions for a specific table</td>
<td>444</td>
</tr>
<tr>
<td>Listing all columns for all tables</td>
<td>445</td>
</tr>
<tr>
<td>Listing or searching columns for a specified table or view</td>
<td>445</td>
</tr>
<tr>
<td>Querying AWS service logs</td>
<td>447</td>
</tr>
<tr>
<td>Application Load Balancer</td>
<td>447</td>
</tr>
<tr>
<td>Elastic Load Balancing</td>
<td>451</td>
</tr>
<tr>
<td>CloudFront</td>
<td>452</td>
</tr>
<tr>
<td>CloudTrail</td>
<td>454</td>
</tr>
<tr>
<td>Amazon EMR</td>
<td>462</td>
</tr>
<tr>
<td>Global Accelerator</td>
<td>465</td>
</tr>
<tr>
<td>GuardDuty</td>
<td>467</td>
</tr>
<tr>
<td>Network Firewall</td>
<td>469</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Amazon Load Balancer</td>
<td>470</td>
</tr>
<tr>
<td>Route 53</td>
<td>472</td>
</tr>
<tr>
<td>Amazon SES</td>
<td>474</td>
</tr>
<tr>
<td>Amazon VPC</td>
<td>474</td>
</tr>
<tr>
<td>AWS WAF</td>
<td>481</td>
</tr>
<tr>
<td>Querying web server logs</td>
<td>489</td>
</tr>
<tr>
<td>Querying Apache logs</td>
<td>490</td>
</tr>
<tr>
<td>Querying IIS logs</td>
<td>492</td>
</tr>
<tr>
<td>Using ACID transactions</td>
<td>498</td>
</tr>
<tr>
<td>Using governed tables</td>
<td>498</td>
</tr>
<tr>
<td>Considerations and limitations</td>
<td>499</td>
</tr>
<tr>
<td>Getting started</td>
<td>499</td>
</tr>
<tr>
<td>Creating governed tables</td>
<td>500</td>
</tr>
<tr>
<td>Querying governed tables</td>
<td>500</td>
</tr>
<tr>
<td>Supported data types</td>
<td>503</td>
</tr>
<tr>
<td>Using iceberg tables</td>
<td>503</td>
</tr>
<tr>
<td>Considerations and limitations</td>
<td>503</td>
</tr>
<tr>
<td>Creating Iceberg tables</td>
<td>504</td>
</tr>
<tr>
<td>Managing Iceberg tables</td>
<td>507</td>
</tr>
<tr>
<td>Evolving Iceberg table schema</td>
<td>510</td>
</tr>
<tr>
<td>Querying Iceberg table data and performing time travel</td>
<td>511</td>
</tr>
<tr>
<td>Updating Iceberg table data</td>
<td>513</td>
</tr>
<tr>
<td>Optimizing Iceberg tables</td>
<td>515</td>
</tr>
<tr>
<td>Supported data types</td>
<td>516</td>
</tr>
<tr>
<td>Other operations</td>
<td>517</td>
</tr>
<tr>
<td>Security</td>
<td>519</td>
</tr>
<tr>
<td>Data protection</td>
<td>519</td>
</tr>
<tr>
<td>Protecting multiple types of data</td>
<td>520</td>
</tr>
<tr>
<td>Encryption at rest</td>
<td>521</td>
</tr>
<tr>
<td>Encryption in transit</td>
<td>529</td>
</tr>
<tr>
<td>Key management</td>
<td>529</td>
</tr>
<tr>
<td>Internetwork traffic privacy</td>
<td>529</td>
</tr>
<tr>
<td>Identity and access management</td>
<td>530</td>
</tr>
<tr>
<td>AWS managed policies</td>
<td>530</td>
</tr>
<tr>
<td>Access through JDBC and ODBC connections</td>
<td>536</td>
</tr>
<tr>
<td>Access to Amazon S3</td>
<td>536</td>
</tr>
<tr>
<td>Cross-account access to S3 buckets</td>
<td>537</td>
</tr>
<tr>
<td>Fine-grained access to databases and tables in AWS Glue</td>
<td>540</td>
</tr>
<tr>
<td>Cross-account access to AWS Glue data catalogs</td>
<td>548</td>
</tr>
<tr>
<td>Access to encrypted metadata in the Data Catalog</td>
<td>553</td>
</tr>
<tr>
<td>Access to workgroups and tags</td>
<td>553</td>
</tr>
<tr>
<td>Allow access to prepared statements</td>
<td>554</td>
</tr>
<tr>
<td>Using CalledVia context keys</td>
<td>555</td>
</tr>
<tr>
<td>Allow access to an Athena Data Connector for External Hive Metastore</td>
<td>557</td>
</tr>
<tr>
<td>Allow Lambda function access to external Hive metastores</td>
<td>559</td>
</tr>
<tr>
<td>Allow access to Athena Federated Query</td>
<td>562</td>
</tr>
<tr>
<td>Allow access to Athena UDF</td>
<td>566</td>
</tr>
<tr>
<td>Allowing access for ML with Athena</td>
<td>570</td>
</tr>
<tr>
<td>Enabling federated access to the Athena API</td>
<td>571</td>
</tr>
<tr>
<td>Logging and monitoring</td>
<td>574</td>
</tr>
<tr>
<td>Logging Amazon Athena API calls with AWS CloudTrail</td>
<td>575</td>
</tr>
<tr>
<td>Compliance validation</td>
<td>577</td>
</tr>
<tr>
<td>Resilience</td>
<td>578</td>
</tr>
<tr>
<td>Infrastructure security</td>
<td>578</td>
</tr>
<tr>
<td>Connect to Amazon Athena using an interface VPC endpoint</td>
<td>579</td>
</tr>
<tr>
<td>Configuration and vulnerability analysis</td>
<td>580</td>
</tr>
<tr>
<td>Using Athena with Lake Formation</td>
<td>580</td>
</tr>
</tbody>
</table>
How Lake Formation data access works ................................................................. 581
Considerations and limitations .............................................................................. 583
Managing user permissions ..................................................................................... 586
Applying Lake Formation permissions to existing databases and tables ................. 588
Using Lake Formation and JDBC or ODBC for federated access .......................... 589
Using workgroups to control query access and costs ............................................ 629
Using workgroups for running queries .................................................................... 629
Benefits of using workgroups .................................................................................. 630
How workgroups work ............................................................................................. 630
Setting up workgroups ............................................................................................. 631
IAM policies for accessing workgroups .................................................................... 632
Workgroup settings .................................................................................................. 638
Managing workgroups ............................................................................................. 639
Athena workgroup APIs .......................................................................................... 645
Troubleshooting workgroups .................................................................................... 645
Controlling costs and monitoring queries with CloudWatch metrics and events .... 647
Video ....................................................................................................................... 647
Enabling CloudWatch query metrics ....................................................................... 647
Monitoring Athena queries with CloudWatch metrics ............................................ 648
Monitoring Athena queries with CloudWatch events .............................................. 651
Monitoring usage metrics ....................................................................................... 654
Setting data usage control limits ............................................................................. 655
Tagging resources ..................................................................................................... 658
Tag basics ................................................................................................................ 658
Tag restrictions ......................................................................................................... 658
Working with tags on workgroups in the console ..................................................... 659
Displaying tags for individual workgroups ............................................................... 659
Adding and deleting tags on an individual workgroup .............................................. 660
Using tag operations ............................................................................................... 660
Managing tags using API operations ....................................................................... 661
Managing tags using the AWS CLI ......................................................................... 662
Tag-based IAM access control policies ................................................................. 664
Tag policy examples for workgroups ....................................................................... 664
Tag policy examples for data catalogs .................................................................... 666
Athena engine versioning ......................................................................................... 669
Changing Athena engine versions ......................................................................... 669
Finding the query engine version for a workgroup ................................................ 669
Changing the engine version using the Athena console .......................................... 670
Changing the engine version using the AWS CLI .................................................... 671
Specifying the engine version when you create a workgroup .................................. 671
Testing queries in advance of an engine version upgrade ....................................... 671
Troubleshooting queries that fail ............................................................................ 672
Athena engine version reference ............................................................................ 672
Athena engine version 3 .......................................................................................... 672
Athena engine version 2 .......................................................................................... 680
SQL reference .......................................................................................................... 690
Data types in Athena ................................................................................................. 690
DML queries, functions, and operators ..................................................................... 692
SELECT .................................................................................................................... 692
INSERT INTO ........................................................................................................... 697
DELETE ................................................................................................................... 700
UPDATE .................................................................................................................. 700
MERGE INTO .......................................................................................................... 701
OPTIMIZE ............................................................................................................... 701
VACUUM .................................................................................................................. 702
EXPLAIN and EXPLAIN ANALYZE .................................................................... 703
PREPARE .................................................................................................................. 715
Considerations and limitations ................................................................. 764

Troubleshooting ....................................................................................... 766

CREATE TABLE AS SELECT (CTAS) ......................................................... 766
Duplicated data occurs with concurrent CTAS statements ...................... 766
HIVE_TOO_MANY_OPEN_PARTITIONS .................................................. 766

Data file issues ......................................................................................... 766
Athena cannot read hidden files .............................................................. 766
Athena cannot read files stored in Amazon S3 Glacier ................................ 767
Athena reads files that I excluded from the AWS Glue crawler ................ 767
HIVE_BAD_DATA: Error parsing field value .............................................. 767
HIVE_CANNOT_OPEN_SPLIT: Error opening Hive split s3://bucket-name .......... 767
HIVE_CURSOR_ERROR: com.amazonaws.services.s3.model.AmazonS3Exception: The specified key does not exist ................................................................. 767
HIVE_CURSOR_ERROR: Unexpected end of input stream ......................... 768
HIVE_FILESYSTEM_ERROR: Incorrect fileSize 1234567 for file ............... 768
HIVE_UNKNOWN_ERROR: Unable to create input format .......................... 768
org.apache.parquet.io.GroupColumnIO cannot be cast to
org.apache.parquet.io.PrimitiveColumnIO .............................................. 768
The S3 location provided to save your query results is invalid. ......... 768

Federated queries ..................................................................................... 768

JSON related errors .................................................................................. 769
NULL or incorrect data errors when trying to read JSON data ................ 769
HIVE_BAD_DATA: Error parsing field value for field 0: java.lang.String cannot be cast to
org.openx.data.jsonserde.json.JSONObject ........................................... 769
HIVE_CURSOR_ERROR: Row is not a valid JSON object - JSONException: Duplicate key .......... 769
HIVE_CURSOR_ERROR messages with pretty-printed JSON ................. 769
Multiple JSON records return a SELECT COUNT of 1 .............................................. 769
Cannot query a table created by a AWS Glue crawler that uses a custom JSON classifier .... 770
MSCK REPAIR TABLE .................................................................................. 770
Output issues ............................................................................................... 770
Unable to verify/create output bucket ......................................................... 770
TIMESTAMP result is empty ........................................................................ 770
Store Athena query output in a format other than CSV .............................. 770
The S3 location provided to save your query results is invalid ................. 770
Partitioning issues ...................................................................................... 771
MSCK REPAIR TABLE does not remove stale partitions ........................... 771
MSCK REPAIR TABLE failure .................................................................... 771
MSCK REPAIR TABLE detects partitions but doesn’t add them to AWS Glue .. 771
Partition projection ranges with the data format of dd-MM-yyyy-HH-mm-ss or yyyy-MM-dd do not work .......................................................... 771
PARTITION BY doesn’t support the BIGINT type ........................................ 771
No meaningful partitions available .............................................................. 771
Partition projection does not work in conjunction with range partitions .... 771
HIVE_UNKNOWN_ERROR: Unable to create input format ............................ 772
HIVE_PARTITION_SCHEMA_MISMATCH .......................................................... 772
SemanticException table is not partitioned but partition spec exists .......... 772
Zero byte _$folder$ files ............................................................................. 772
Zero records returned from partitioned data ............................................. 772
Permissions .................................................................................................. 772
Access denied error when querying Amazon S3 ........................................ 772
Access denied with status code: 403 error when running DDL queries on encrypted data in Amazon S3 ................................................................. 773
Access denied with status code: 403 when querying an Amazon S3 bucket in another account ... 773
Use IAM role credentials to connect to the Athena JDBC driver ............... 774
Query syntax issues .................................................................................... 774
FAILED: NullPointerException name is null ............................................ 774
Function not registered ............................................................................. 774
GENERIC_INTERNAL_ERROR exceptions ...................................................... 774
Number of matching groups doesn’t match the number of columns ........ 775
queryString failed to satisfy constraint: Member must have length less than or equal to 262144 775
SYNTAX_ERROR: Column cannot be resolved ............................................. 775
Throttling issues .......................................................................................... 775
Views ........................................................................................................... 776
Views created in Apache Hive shell do not work in Athena ................. 776
View is stale; it must be re-created ............................................................ 776
Workgroups ............................................................................................... 776
Additional resources .................................................................................. 776
Athena error catalog .................................................................................. 777
Error category ............................................................................................ 777
Error type reference ................................................................................... 777
Performance tuning ..................................................................................... 781
Physical limits ............................................................................................. 781
Query optimization techniques ................................................................. 781
Data size .................................................................................................... 781
File formats ............................................................................................... 782
Joins, grouping, and unions ................................................................. 782
Partitioning .............................................................................................. 783
Window functions .................................................................................... 783
Use more efficient functions ................................................................. 783
Additional resources ................................................................................ 784
Service Quotas .......................................................................................... 785
Queries ..................................................................................................... 785
Query string length .................................................................................. 785
<table>
<thead>
<tr>
<th>Workgroups</th>
<th>785</th>
</tr>
</thead>
<tbody>
<tr>
<td>Databases, tables, and partitions</td>
<td>786</td>
</tr>
<tr>
<td>Amazon S3 buckets</td>
<td>786</td>
</tr>
<tr>
<td>Per account API call quotas</td>
<td>786</td>
</tr>
<tr>
<td>Code samples</td>
<td>788</td>
</tr>
<tr>
<td>Constants</td>
<td>788</td>
</tr>
<tr>
<td>Create a client to access Athena</td>
<td>788</td>
</tr>
<tr>
<td>Start query execution</td>
<td>789</td>
</tr>
<tr>
<td>Stop query execution</td>
<td>791</td>
</tr>
<tr>
<td>List query executions</td>
<td>793</td>
</tr>
<tr>
<td>Create a named query</td>
<td>794</td>
</tr>
<tr>
<td>Delete a named query</td>
<td>795</td>
</tr>
<tr>
<td>List named queries</td>
<td>796</td>
</tr>
<tr>
<td>Release notes</td>
<td>798</td>
</tr>
<tr>
<td>2022</td>
<td>798</td>
</tr>
<tr>
<td>November 18, 2022</td>
<td>798</td>
</tr>
<tr>
<td>November 17, 2022</td>
<td>798</td>
</tr>
<tr>
<td>November 14, 2022</td>
<td>799</td>
</tr>
<tr>
<td>November 11, 2022</td>
<td>799</td>
</tr>
<tr>
<td>November 8, 2022</td>
<td>800</td>
</tr>
<tr>
<td>October 13, 2022</td>
<td>800</td>
</tr>
<tr>
<td>October 10, 2022</td>
<td>800</td>
</tr>
<tr>
<td>September 23, 2022</td>
<td>800</td>
</tr>
<tr>
<td>September 13, 2022</td>
<td>801</td>
</tr>
<tr>
<td>August 31, 2022</td>
<td>801</td>
</tr>
<tr>
<td>August 23, 2022</td>
<td>801</td>
</tr>
<tr>
<td>August 3, 2022</td>
<td>801</td>
</tr>
<tr>
<td>August 1, 2022</td>
<td>802</td>
</tr>
<tr>
<td>July 21, 2022</td>
<td>802</td>
</tr>
<tr>
<td>July 11, 2022</td>
<td>803</td>
</tr>
<tr>
<td>July 8, 2022</td>
<td>803</td>
</tr>
<tr>
<td>June 6, 2022</td>
<td>803</td>
</tr>
<tr>
<td>May 25, 2022</td>
<td>803</td>
</tr>
<tr>
<td>May 6, 2022</td>
<td>804</td>
</tr>
<tr>
<td>April 22, 2022</td>
<td>804</td>
</tr>
<tr>
<td>April 21, 2022</td>
<td>804</td>
</tr>
<tr>
<td>April 13, 2022</td>
<td>805</td>
</tr>
<tr>
<td>March 30, 2022</td>
<td>805</td>
</tr>
<tr>
<td>March 18, 2022</td>
<td>805</td>
</tr>
<tr>
<td>March 2, 2022</td>
<td>806</td>
</tr>
<tr>
<td>February 23, 2022</td>
<td>806</td>
</tr>
<tr>
<td>February 15, 2022</td>
<td>806</td>
</tr>
<tr>
<td>February 14, 2022</td>
<td>807</td>
</tr>
<tr>
<td>February 9, 2022</td>
<td>807</td>
</tr>
<tr>
<td>February 8, 2022</td>
<td>807</td>
</tr>
<tr>
<td>January 28, 2022</td>
<td>807</td>
</tr>
<tr>
<td>January 13, 2022</td>
<td>808</td>
</tr>
<tr>
<td>2021</td>
<td>808</td>
</tr>
<tr>
<td>November 30, 2021</td>
<td>808</td>
</tr>
<tr>
<td>November 26, 2021</td>
<td>808</td>
</tr>
<tr>
<td>November 24, 2021</td>
<td>809</td>
</tr>
<tr>
<td>November 22, 2021</td>
<td>809</td>
</tr>
<tr>
<td>November 18, 2021</td>
<td>809</td>
</tr>
<tr>
<td>November 17, 2021</td>
<td>810</td>
</tr>
<tr>
<td>November 16, 2021</td>
<td>810</td>
</tr>
<tr>
<td>November 12, 2021</td>
<td>811</td>
</tr>
<tr>
<td>November 2, 2021</td>
<td>811</td>
</tr>
<tr>
<td>Date</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------</td>
<td>------</td>
</tr>
<tr>
<td>October 29, 2021</td>
<td>811</td>
</tr>
<tr>
<td>October 4, 2021</td>
<td>812</td>
</tr>
<tr>
<td>September 16, 2021</td>
<td>812</td>
</tr>
<tr>
<td>September 15, 2021</td>
<td>813</td>
</tr>
<tr>
<td>August 31, 2021</td>
<td>813</td>
</tr>
<tr>
<td>August 12, 2021</td>
<td>813</td>
</tr>
<tr>
<td>August 6, 2021</td>
<td>814</td>
</tr>
<tr>
<td>August 5, 2021</td>
<td>814</td>
</tr>
<tr>
<td>July 30, 2021</td>
<td>814</td>
</tr>
<tr>
<td>July 21, 2021</td>
<td>814</td>
</tr>
<tr>
<td>July 16, 2021</td>
<td>815</td>
</tr>
<tr>
<td>July 8, 2021</td>
<td>815</td>
</tr>
<tr>
<td>July 1, 2021</td>
<td>815</td>
</tr>
<tr>
<td>June 23, 2021</td>
<td>815</td>
</tr>
<tr>
<td>May 12, 2021</td>
<td>816</td>
</tr>
<tr>
<td>May 10, 2021</td>
<td>816</td>
</tr>
<tr>
<td>May 5, 2021</td>
<td>816</td>
</tr>
<tr>
<td>April 30, 2021</td>
<td>816</td>
</tr>
<tr>
<td>April 29, 2021</td>
<td>817</td>
</tr>
<tr>
<td>April 26, 2021</td>
<td>817</td>
</tr>
<tr>
<td>April 21, 2021</td>
<td>817</td>
</tr>
<tr>
<td>April 5, 2021</td>
<td>817</td>
</tr>
<tr>
<td>March 30, 2021</td>
<td>818</td>
</tr>
<tr>
<td>March 25, 2021</td>
<td>818</td>
</tr>
<tr>
<td>March 5, 2021</td>
<td>818</td>
</tr>
<tr>
<td>February 25, 2021</td>
<td>818</td>
</tr>
<tr>
<td>2020</td>
<td></td>
</tr>
<tr>
<td>December 16, 2020</td>
<td>818</td>
</tr>
<tr>
<td>November 24, 2020</td>
<td>819</td>
</tr>
<tr>
<td>November 11, 2020</td>
<td>819</td>
</tr>
<tr>
<td>October 22, 2020</td>
<td>820</td>
</tr>
<tr>
<td>July 29, 2020</td>
<td>820</td>
</tr>
<tr>
<td>July 9, 2020</td>
<td>821</td>
</tr>
<tr>
<td>June 1, 2020</td>
<td>821</td>
</tr>
<tr>
<td>May 21, 2020</td>
<td>821</td>
</tr>
<tr>
<td>April 1, 2020</td>
<td>821</td>
</tr>
<tr>
<td>March 11, 2020</td>
<td>822</td>
</tr>
<tr>
<td>March 6, 2020</td>
<td>822</td>
</tr>
<tr>
<td>2019</td>
<td></td>
</tr>
<tr>
<td>November 26, 2019</td>
<td>822</td>
</tr>
<tr>
<td>November 12, 2019</td>
<td>825</td>
</tr>
<tr>
<td>November 8, 2019</td>
<td>825</td>
</tr>
<tr>
<td>October 8, 2019</td>
<td>825</td>
</tr>
<tr>
<td>September 19, 2019</td>
<td>825</td>
</tr>
<tr>
<td>September 12, 2019</td>
<td>826</td>
</tr>
<tr>
<td>August 16, 2019</td>
<td>826</td>
</tr>
<tr>
<td>August 9, 2019</td>
<td>826</td>
</tr>
<tr>
<td>June 26, 2019</td>
<td>826</td>
</tr>
<tr>
<td>May 24, 2019</td>
<td>826</td>
</tr>
<tr>
<td>March 05, 2019</td>
<td>826</td>
</tr>
<tr>
<td>February 22, 2019</td>
<td>827</td>
</tr>
<tr>
<td>February 18, 2019</td>
<td>827</td>
</tr>
<tr>
<td>2018</td>
<td></td>
</tr>
<tr>
<td>November 20, 2018</td>
<td>829</td>
</tr>
<tr>
<td>October 15, 2018</td>
<td>829</td>
</tr>
<tr>
<td>October 10, 2018</td>
<td>829</td>
</tr>
<tr>
<td>September 6, 2018</td>
<td>830</td>
</tr>
<tr>
<td>Date</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------</td>
</tr>
<tr>
<td>August 23, 2018</td>
<td>830</td>
</tr>
<tr>
<td>August 16, 2018</td>
<td>831</td>
</tr>
<tr>
<td>August 7, 2018</td>
<td>831</td>
</tr>
<tr>
<td>June 5, 2018</td>
<td>831</td>
</tr>
<tr>
<td>May 17, 2018</td>
<td>832</td>
</tr>
<tr>
<td>April 19, 2018</td>
<td>833</td>
</tr>
<tr>
<td>April 6, 2018</td>
<td>833</td>
</tr>
<tr>
<td>March 15, 2018</td>
<td>833</td>
</tr>
<tr>
<td>February 2, 2018</td>
<td>833</td>
</tr>
<tr>
<td>January 19, 2018</td>
<td>833</td>
</tr>
<tr>
<td>2017</td>
<td>834</td>
</tr>
<tr>
<td>November 13, 2017</td>
<td>834</td>
</tr>
<tr>
<td>November 1, 2017</td>
<td>834</td>
</tr>
<tr>
<td>October 19, 2017</td>
<td>834</td>
</tr>
<tr>
<td>October 3, 2017</td>
<td>834</td>
</tr>
<tr>
<td>September 25, 2017</td>
<td>835</td>
</tr>
<tr>
<td>August 14, 2017</td>
<td>835</td>
</tr>
<tr>
<td>August 4, 2017</td>
<td>835</td>
</tr>
<tr>
<td>June 22, 2017</td>
<td>835</td>
</tr>
<tr>
<td>June 8, 2017</td>
<td>835</td>
</tr>
<tr>
<td>May 19, 2017</td>
<td>835</td>
</tr>
<tr>
<td>April 4, 2017</td>
<td>836</td>
</tr>
<tr>
<td>March 24, 2017</td>
<td>837</td>
</tr>
<tr>
<td>February 20, 2017</td>
<td>838</td>
</tr>
</tbody>
</table>

Document history .............................................................................. 840
AWS glossary ..................................................................................... 851
What is Amazon Athena?

Amazon Athena is an interactive query service that makes it easy to analyze data directly in Amazon Simple Storage Service (Amazon S3) using standard SQL (p. 690). With a few actions in the AWS Management Console, you can point Athena at your data stored in Amazon S3 and begin using standard SQL to run ad-hoc queries and get results in seconds.

Athena is serverless, so there is no infrastructure to set up or manage, and you pay only for the queries you run. Athena scales automatically—running queries in parallel—so results are fast, even with large datasets and complex queries.

Topics

- When should I use Athena? (p. 1)
- Accessing Athena (p. 2)
- Understanding tables, databases, and the Data Catalog (p. 3)
- AWS service integrations with Athena (p. 4)

When should I use Athena?

Query services like Amazon Athena, data warehouses like Amazon Redshift, and sophisticated data processing frameworks like Amazon EMR all address different needs and use cases. The following guidance can help you choose one or more services based on your requirements.

Amazon Athena

Athena helps you analyze unstructured, semi-structured, and structured data stored in Amazon S3. Examples include CSV, JSON, or columnar data formats such as Apache Parquet and Apache ORC. You can use Athena to run ad-hoc queries using ANSI SQL, without the need to aggregate or load the data into Athena.

Athena integrates with Amazon QuickSight for easy data visualization. You can use Athena to generate reports or to explore data with business intelligence tools or SQL clients connected with a JDBC or an ODBC driver. For more information, see What is Amazon QuickSight in the Amazon QuickSight User Guide and Connecting to Amazon Athena with ODBC and JDBC drivers (p. 183).

Athena integrates with the AWS Glue Data Catalog, which offers a persistent metadata store for your data in Amazon S3. This allows you to create tables and query data in Athena based on a central metadata store available throughout your Amazon Web Services account and integrated with the ETL and data discovery features of AWS Glue. For more information, see Integration with AWS Glue (p. 21) and What is AWS Glue in the AWS Glue Developer Guide.

Amazon Athena makes it easy to run interactive queries against data directly in Amazon S3 without having to format data or manage infrastructure. For example, Athena is useful if you want to run a quick query on web logs to troubleshoot a performance issue on your site. With Athena, you can get started fast: you just define a table for your data and start querying using standard SQL.

You should use Amazon Athena if you want to run interactive ad hoc SQL queries against data on Amazon S3, without having to manage any infrastructure or clusters. Amazon Athena provides the easiest way to run ad hoc queries for data in Amazon S3 without the need to setup or manage any servers.
For a list of AWS services that Athena leverages or integrates with, see the section called “AWS service integrations with Athena” (p. 4).

Amazon EMR

Amazon EMR makes it simple and cost effective to run highly distributed processing frameworks such as Hadoop, Spark, and Presto when compared to on-premises deployments. Amazon EMR is flexible – you can run custom applications and code, and define specific compute, memory, storage, and application parameters to optimize your analytic requirements.

In addition to running SQL queries, Amazon EMR can run a wide variety of scale-out data processing tasks for applications such as machine learning, graph analytics, data transformation, streaming data, and virtually anything you can code. You should use Amazon EMR if you use custom code to process and analyze extremely large datasets with the latest big data processing frameworks such as Spark, Hadoop, Presto, or Hbase. Amazon EMR gives you full control over the configuration of your clusters and the software installed on them.

You can use Amazon Athena to query data that you process using Amazon EMR. Amazon Athena supports many of the same data formats as Amazon EMR. Athena's data catalog is Hive metastore compatible. If you use EMR and already have a Hive metastore, you can run your DDL statements on Amazon Athena and query your data immediately without affecting your Amazon EMR jobs.

Amazon Redshift

A data warehouse like Amazon Redshift is your best choice when you need to pull together data from many different sources – like inventory systems, financial systems, and retail sales systems – into a common format, and store it for long periods of time. If you want to build sophisticated business reports from historical data, then a data warehouse like Amazon Redshift is the best choice. The query engine in Amazon Redshift has been optimized to perform especially well on running complex queries that join large numbers of very large database tables. When you need to run queries against highly structured data with lots of joins across lots of very large tables, choose Amazon Redshift.

For more information about when to use Athena, consult the following resources:

- When to use Athena vs other big data services
- Amazon Athena overview
- Amazon Athena features
- Amazon Athena FAQs
- Amazon Athena blog posts

Accessing Athena

You can access Athena using the AWS Management Console, a JDBC or ODBC connection, the Athena API, the Athena CLI, the AWS SDK, or AWS Tools for Windows PowerShell.

- To get started with the console, see Getting Started (p. 9).
- To learn how to use JDBC or ODBC drivers, see Using Athena with the JDBC driver (p. 183) and Connecting to Amazon Athena with ODBC (p. 185).
- To use the Athena API, see the Amazon Athena API Reference.
- To use the CLI, install the AWS CLI and then type aws athena help from the command line to see available commands. For information about available commands, see the Amazon Athena command line reference.
To use the AWS SDK for Java 2.x, see the Athena section of the AWS SDK for Java 2.x API Reference, the Athena Java V2 examples on GitHub.com, and the AWS SDK for Java 2.x Developer Guide.

To use the AWS SDK for .NET, see the Amazon.Athena namespace in the AWS SDK for .NET API Reference, the .NET Athena examples on GitHub.com, and the AWS SDK for .NET Developer Guide.


For information about Athena service endpoints that you can connect to programmatically, see Amazon Athena endpoints and quotas in the Amazon Web Services General Reference.

Understanding tables, databases, and the Data Catalog

In Athena, tables and databases are containers for the metadata definitions that define a schema for underlying source data. For each dataset, a table needs to exist in Athena. The metadata in the table tells Athena where the data is located in Amazon S3, and specifies the structure of the data, for example, column names, data types, and the name of the table. Databases are a logical grouping of tables, and also hold only metadata and schema information for a dataset.

For each dataset that you'd like to query, Athena must have an underlying table it will use for obtaining and returning query results. Therefore, before querying data, a table must be registered in Athena. The registration occurs when you either create tables automatically or manually.

Regardless of how the tables are created, the tables creation process registers the dataset with Athena. This registration occurs in the AWS Glue Data Catalog and enables Athena to run queries on the data.

To create a table automatically, use an AWS Glue crawler from within Athena. For more information about AWS Glue and crawlers, see Integration with AWS Glue (p. 21). When AWS Glue creates a table, it registers it in its own AWS Glue Data Catalog. Athena uses the AWS Glue Data Catalog to store and retrieve this metadata, using it when you run queries to analyze the underlying dataset.

After you create a table, you can use SQL SELECT (p. 692) statements to query it, including getting specific file locations for your source data (p. 696). Your query results are stored in Amazon S3 in the query result location that you specify (p. 344).

The AWS Glue Data Catalog is accessible throughout your Amazon Web Services account. Other AWS services can share the AWS Glue Data Catalog, so you can see databases and tables created throughout your organization using Athena and vice versa. In addition, AWS Glue lets you automatically discover data schema and extract, transform, and load (ETL) data.

To create a table manually:
- Use the Athena console to run the Create Table Wizard.
- Use the Athena console to write Hive DDL statements in the Query Editor.
- Use the Athena API or CLI to run a SQL query string with DDL statements.
- Use the Athena JDBC or ODBC driver.

When you create tables and databases manually, Athena uses HiveQL data definition language (DDL) statements such as CREATE TABLE, CREATE DATABASE, and DROP TABLE under the hood to create tables and databases in the AWS Glue Data Catalog.

Note
If you have tables in Athena created before August 14, 2017, they were created in an Athena-managed internal data catalog that exists side-by-side with the AWS Glue Data Catalog until
you choose to update. For more information, see Upgrading to the AWS Glue Data Catalog step-by-step (p. 31).

When you query an existing table, under the hood, Amazon Athena uses Presto, a distributed SQL engine. We have examples with sample data within Athena to show you how to create a table and then issue a query against it using Athena. Athena also has a tutorial in the console that helps you get started creating a table based on data that is stored in Amazon S3.

• For a step-by-step tutorial on creating a table and writing queries in the Athena Query Editor, see Getting started (p. 9).
• Run the Athena tutorial in the console. This launches automatically if you log in to https://console.aws.amazon.com/athena/ for the first time. You can also choose Tutorial in the console to launch it.

AWS service integrations with Athena

You can use Athena to query data from the AWS services listed in this section. To see the Regions that each service supports, see Regions and endpoints in the Amazon Web Services General Reference.

AWS services integrated with Athena

• AWS CloudFormation
• Amazon CloudFront
• AWS CloudTrail
• Elastic Load Balancing
• AWS Glue Data Catalog
• AWS Identity and Access Management (IAM)
• Amazon QuickSight
• Amazon S3 Inventory
• AWS Step Functions
• AWS Systems Manager Inventory
• Amazon Virtual Private Cloud

For information about each integration, see the following sections.

AWS CloudFormation

Data Catalog

Reference topic: AWS::Athena::DataCatalog in the AWS CloudFormation User Guide

Specify an Athena data catalog, including a name, description, type, parameters, and tags. For more information, see DataCatalog in the Amazon Athena API Reference.

Named Query

Reference topic: AWS::Athena::NamedQuery in the AWS CloudFormation User Guide

Specify named queries with AWS CloudFormation and run them in Athena. Named queries allow you to map a query name to a query and then run it as a saved query from the Athena console. For information, see CreateNamedQuery in the Amazon Athena API Reference.

Workgroup

Reference topic: AWS::Athena::WorkGroup in the AWS CloudFormation User Guide
Specify Athena workgroups using AWS CloudFormation. Use Athena workgroups to isolate queries for you or your group from other queries in the same account. For more information, see Using workgroups to control query access and costs (p. 629) in the Amazon Athena User Guide and CreateWorkGroup in the Amazon Athena API Reference.

**Amazon CloudFront**

Reference topic: Querying Amazon CloudFront logs (p. 452)

Use Athena to query Amazon CloudFront logs. For more information about using CloudFront, see the Amazon CloudFront Developer Guide.

**AWS CloudTrail**

Reference topic: Querying AWS CloudTrail logs (p. 454)

Using Athena with CloudTrail logs is a powerful way to enhance your analysis of AWS service activity. For example, you can use queries to identify trends and further isolate activity by attribute, such as source IP address or user. You can create tables for querying logs directly from the CloudTrail console, and use those tables to run queries in Athena. For more information, see Using the CloudTrail console to create an Athena table for CloudTrail logs (p. 456).

**Elastic Load Balancing**

Reference topic: Querying Application Load Balancer logs (p. 447)

Querying Application Load Balancer logs allows you to see the source of traffic, latency, and bytes transferred to and from Elastic Load Balancing instances and backend applications. For more information, see Creating the table for ALB logs (p. 448).

Reference topic: Querying Classic Load Balancer logs (p. 451)

Query Classic Load Balancer logs to analyze and understand traffic patterns to and from Elastic Load Balancing instances and backend applications. You can see the source of traffic, latency, and bytes transferred. For more information, see Creating the Table for ELB Logs (p. 451).

**AWS Glue Data Catalog**

Reference topic: Integration with AWS Glue (p. 21)

Athena integrates with the AWS Glue Data Catalog, which offers a persistent metadata store for your data in Amazon S3. This allows you to create tables and query data in Athena based on a central metadata store available throughout your Amazon Web Services account and integrated with the ETL and data discovery features of AWS Glue. For more information, see Integration with AWS Glue (p. 21) and What is AWS Glue in the AWS Glue Developer Guide.

**AWS Identity and Access Management (IAM)**

Reference topic: Actions for Amazon Athena

You can use Athena API actions in IAM permission policies. For more information, see Actions for Amazon Athena and Identity and access management in Athena (p. 530).

**Amazon QuickSight**

Reference topic: Connecting to Amazon Athena with ODBC and JDBC drivers (p. 183)

Athena integrates with Amazon QuickSight for easy data visualization. You can use Athena to generate reports or to explore data with business intelligence tools or SQL clients connected with a JDBC or an ODBC driver. For more information about Amazon QuickSight, see What is Amazon QuickSight in the Amazon QuickSight User Guide. For information about using JDBC and ODBC drivers with Athena, see Connecting to Amazon Athena with ODBC and JDBC Drivers (p. 183).

**Amazon S3 Inventory**

Reference topic: Querying inventory with Athena in the Amazon Simple Storage Service User Guide
You can use Amazon Athena to query Amazon S3 inventory using standard SQL. You can use Amazon S3 inventory to audit and report on the replication and encryption status of your objects for business, compliance, and regulatory needs. For more information, see Amazon S3 inventory in the Amazon Simple Storage Service User Guide.

**AWS Step Functions**

Reference topic: Call Athena with Step Functions in the AWS Step Functions Developer Guide

Call Athena with AWS Step Functions. AWS Step Functions can control select AWS services directly using the Amazon States Language. You can use Step Functions with Athena to start and stop query execution, get query results, run ad-hoc or scheduled data queries, and retrieve results from data lakes in Amazon S3. For more information, see the AWS Step Functions Developer Guide.

**Video: Orchestrate Amazon Athena Queries using AWS Step Functions**

The following video demonstrates how to use Amazon Athena and AWS Step Functions to run a regularly scheduled Athena query and generate a corresponding report.

Orchestrate Amazon Athena queries using AWS Step Functions

For an example that uses Step Functions and Amazon EventBridge to orchestrate AWS Glue DataBrew, Athena, and Amazon QuickSight, see Orchestrating an AWS Glue DataBrew job and Amazon Athena query with AWS Step Functions in the AWS Big Data Blog.

**AWS Systems Manager Inventory**

Reference topic: Querying inventory data from multiple regions and accounts in the AWS Systems Manager User Guide

AWS Systems Manager Inventory integrates with Amazon Athena to help you query inventory data from multiple AWS Regions and accounts. For more information, see the AWS Systems Manager User Guide.

**Amazon Virtual Private Cloud**

Reference topic: Querying Amazon VPC flow logs (p. 474)

Amazon Virtual Private Cloud flow logs capture information about the IP traffic going to and from network interfaces in a VPC. Query the logs in Athena to investigate network traffic patterns and identify threats and risks across your Amazon VPC network. For more information about Amazon VPC, see the Amazon VPC User Guide.
Setting up

If you've already signed up for Amazon Web Services, you can start using Amazon Athena immediately. If you haven't signed up for AWS or need assistance getting started, be sure to complete the following tasks:

1. Sign up for an AWS account (p. 7)
2. Create an IAM administrator user and group (p. 7)
3. Attach managed policies for Athena (p. 7)
4. Sign in as an IAM user (p. 8)

1. Sign up for an AWS account

When you sign up for AWS, your account is automatically signed up for all services in AWS, including Athena. You are charged only for the services that you use. For pricing information, see Amazon Athena pricing.

If you have an AWS account already, skip to the next task. If you don't have an AWS account, use the following procedure to create one.

To create an AWS account

1. Open http://aws.amazon.com/, and then choose Create an AWS account.
2. Follow the online instructions. Part of the sign-up procedure involves receiving a phone call and entering a PIN using the phone keypad.
3. Note your AWS account number, because you need it for the next task.

2. Create an IAM administrator user and group

An AWS Identity and Access Management (IAM) user is an account that you create to access services. It is a different user than your main AWS account. As a security best practice, we recommend that you use the IAM user's credentials to access AWS services. You use the IAM console to create an administrator IAM user and an Administrators group for the user. You can then access the console for Athena and other AWS services by accessing a special link and providing the credentials for the IAM user that you created.

For steps, see Creating an administrator IAM user and user group (console) in the IAM User Guide.

3. Attach managed policies for Athena

After you have created an IAM user, you must attach some Athena managed policies to the user so that the user can access Athena. There are two managed policies for Athena: AmazonAthenaFullAccess and AWSQuicksightAthenaAccess. These policies grant permissions to Athena to query Amazon S3 and to write the results of your queries to a separate bucket on your behalf. To see the contents of these policies for Athena, see AWS managed policies for Amazon Athena (p. 530).
4. Sign in as an IAM user

For steps to attach the Athena managed policies, follow Adding IAM identity permissions (console) in the IAM User Guide and add the AmazonAthenaFullAccess and AWSThirdPartyAthenaAccess managed policies to the IAM administrator user that you created.

**Note**
You may need additional permissions to access the underlying dataset in Amazon S3. If you are not the account owner or otherwise have restricted access to a bucket, contact the bucket owner to grant access using a resource-based bucket policy, or contact your account administrator to grant access using an identity-based policy. For more information, see Access to Amazon S3 (p. 536). If the dataset or Athena query results are encrypted, you may need additional permissions. For more information, see Encryption at rest (p. 521).

4. Sign in as an IAM user

To sign in as the new IAM user that you created, you can use the custom sign-in URL for the IAM users of your account. To see the sign-in URL for the IAM users for your account, open the IAM console and choose Users, **user_name**, Security credentials, **Console sign-in link**. As a convenience, you can use the clipboard icon to copy the sign-in URL to the clipboard.

For more information about signing in as an IAM user, see How IAM users sign in to your AWS account in the IAM User Guide.
Getting started

This tutorial walks you through using Amazon Athena to query data. You'll create a table based on sample data stored in Amazon Simple Storage Service, query the table, and check the results of the query.

The tutorial uses live resources, so you are charged for the queries that you run. You aren't charged for the sample data in the location that this tutorial uses, but if you upload your own data files to Amazon S3, charges do apply.

Prerequisites

- If you have not already done so, sign up for an account in Setting up (p. 7).
- Using the same AWS Region (for example, US West (Oregon)) and account that you are using for Athena, Create a bucket in Amazon S3 to hold your Athena query results.

Step 1: Create a database

You first need to create a database in Athena.

To create an Athena database

2. If this is your first time to visit the Athena console in your current AWS Region, choose Explore the query editor to open the query editor. Otherwise, Athena opens in the query editor.
3. Choose View Settings to set up a query result location in Amazon S3.
4. On the Settings tab, choose Manage.
5. For Manage settings, do one of the following:

- In the Location of query result box, enter the path to the bucket that you created in Amazon S3 for your query results. Prefix the path with `s3://`.  
- Choose Browse S3, choose the Amazon S3 bucket that you created for your current Region, and then choose Choose.
6. Choose **Save**.

7. Choose **Editor** to switch to the query editor.

8. On the right of the navigation pane, you can use the Athena query editor to enter and run queries and statements.
9. To create a database named mydatabase, enter the following CREATE DATABASE statement.

```
CREATE DATABASE mydatabase
```

10. Choose Run or press Ctrl+ENTER.

11. From the Database list on the left, choose mydatabase to make it your current database.

---

**Step 2: Create a table**

Now that you have a database, you can create an Athena table for it. The table that you create will be based on sample Amazon CloudFront log data in the location s3://athena-examples-myregion/cloudfront/plaintext/, where myregion is your current AWS Region.

The sample log data is in tab-separated values (TSV) format, which means that a tab character is used as a delimiter to separate the fields. The data looks like the following example. For readability, the tabs in the excerpt have been converted to spaces and the final field shortened.

```
2014-07-05 20:00:09 DFW3 4260 10.0.0.15 GET eabcd12345678.cloudfront.net /test-image-1.jpeg 200 - Mozilla/5.0[...]
2014-07-05 20:00:09 DFW3 4252 10.0.0.15 GET eabcd12345678.cloudfront.net /test-image-2.jpeg 200 - Mozilla/5.0[...]
2014-07-05 20:00:10 AMS1 4261 10.0.0.15 GET eabcd12345678.cloudfront.net /test-image-3.jpeg 200 - Mozilla/5.0[...]
```

To enable Athena to read this data, you could run a CREATE EXTERNAL TABLE statement like the following. The statement that creates the table defines columns that map to the data, specifies how the data is delimited, and specifies the Amazon S3 location that contains the sample data.

**Note**

For the LOCATION clause, specify an Amazon S3 folder location, not a specific file. Athena scans all of the files in the folder that you specify.
CREATE EXTERNAL TABLE IF NOT EXISTS cloudfront_logs (  `Date` DATE,  Time STRING,  Location STRING,  Bytes INT,  RequestIP STRING,  Method STRING,  Host STRING,  Uri STRING,  Status INT,  Referrer STRING,  ClientInfo STRING  )  ROW FORMAT DELIMITED  FIELDS TERMINATED BY '	'  LINES TERMINATED BY '
'  LOCATION 's3://athena-examples-my-region/cloudfront/plaintext/';

The example creates a table called cloudfront_logs and specifies a name and data type for each field. These fields become the columns in the table. Because date is a reserved word (p. 256), it is escaped with backtick (`) characters. ROW FORMAT DELIMITED means that Athena will use a default library called LazySimpleSerDe (p. 324) to do the actual work of parsing the data. The example also specifies that the fields are tab separated (FIELDS TERMINATED BY '	') and that each record in the file ends in a newline character (LINES TERMINATED BY '
'). Finally, the LOCATION clause specifies the path in Amazon S3 where the actual data to be read is located. If you have your own tab or comma-separated data, you can use a CREATE TABLE statement like this.

Returning to the sample data, here is a full example of the final field ClientInfo:

Mozilla/5.0%20(Android;%20U;%20Windows%20NT%205.1;%20en-US;%20rv:1.9.0.9)%20Gecko/2009040821%20IE/3.0.9

As you can see, this field is multivalued. If the CREATE TABLE statement specifies tabs as field delimiters, the separate components inside this particular field can't be broken out into separate columns. To create columns from the values inside the field, you can use a regular expression (regex) that contains regex groups. The regex groups that you specify become separate table columns. To use a regex in your CREATE TABLE statement, use syntax like the following. This syntax instructs Athena to use the Regex SerDe (p. 337) library and the regular expression that you specify.

ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'  WITH SERDEPROPERTIES ("input.regex" = "regular_expression")

Regular expressions can be useful for creating tables from complex CSV or TSV data but can be difficult to write and maintain. Fortunately, there are other libraries that you can use for formats like JSON, Parquet, and ORC. For more information, see Supported SerDes and data formats (p. 303).

Now you are ready to create the table in the Athena query editor. The CREATE TABLE statement and regex are provided for you.

To create a table in Athena

1. In the navigation pane, for Database, make sure that mydatabase is selected.
2. To give yourself more room in the query editor, you can choose the arrow icon to collapse the navigation pane.
3. To create a tab for a new query, choose the plus (+) sign in the query editor. You can have up to ten query tabs open at once.

4. To close one or more query tabs, choose the arrow next to the plus sign. To close all tabs at once, choose the arrow, and then choose **Close all tabs.**
Amazon Athena User Guide
Step 2: Create a table

5.

In the query pane, enter the following CREATE EXTERNAL TABLE statement. The regex breaks out
the operating system, browser, and browser version information from the ClientInfo ﬁeld in the
log data.
CREATE EXTERNAL TABLE IF NOT EXISTS cloudfront_logs (
`Date` DATE,
Time STRING,
Location STRING,
Bytes INT,
RequestIP STRING,
Method STRING,
Host STRING,
Uri STRING,
Status INT,
Referrer STRING,
os STRING,
Browser STRING,
BrowserVersion STRING
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
"input.regex" = "^(?!#)([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s
+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s+[^\(]+[\(]([^\;]+).*\%20([^
\/]+)[\/](.*)$"
) LOCATION 's3://athena-examples-myregion/cloudfront/plaintext/';

6.

In the LOCATION statement, replace myregion with the AWS Region that you are currently using
(for example, us-west-1).

7.

Choose Run.

15


The table `cloudfront_logs` is created and appears under the list of Tables for the mydatabase database.

**Step 3: Query data**

Now that you have the `cloudfront_logs` table created in Athena based on the data in Amazon S3, you can run SQL queries on the table and see the results in Athena. For more information about using SQL in Athena, see SQL reference for Amazon Athena (p. 690).

**To run a query**

1. Choose the plus (+) sign to open a new query tab and enter the following SQL statement in the query pane.

   ```sql
   SELECT os, COUNT(*) count
   FROM cloudfront_logs
   WHERE date BETWEEN date '2014-07-05' AND date '2014-08-05'
   GROUP BY os
   ```

2. Choose Run.

   The results look like the following:

   ![Query Results](image)

3. To save the results of the query to a .csv file, choose Download results.
4. To view or run previous queries, choose the **Recent queries** tab.

5. To download the results of a previous query from the **Recent queries** tab, select the query, and then choose **Download results**. Queries are retained for 45 days.
Saving your queries

You can save the queries that you create or edit in the query editor with a name. Athena stores these queries on the Saved queries tab. You can use the Saved queries tab to recall, run, rename, or delete your saved queries. For more information, see Using saved queries (p. 368).

Connecting to other data sources

This tutorial used a data source in Amazon S3 in CSV format. For information about using Athena with AWS Glue, see Using AWS Glue to connect to data sources in Amazon S3 (p. 22). You can also connect Athena to a variety of data sources by using ODBC and JDBC drivers, external Hive metastores, and Athena data source connectors. For more information, see Connecting to data sources (p. 21).
Accessing Amazon Athena

You can access Amazon Athena using the AWS Management Console, the Amazon Athena API, or the AWS CLI. For information about IAM policies in Athena, see Identity and access management in Athena (p. 530).

Using the console

You can use the AWS Management Console for Amazon Athena to do the following:

- Create or select a database.
- Create, view, and delete tables.
- Filter tables by starting to type their names.
- Preview tables and generate CREATE TABLE DDL for them.
- Show table properties.
- Run queries on tables, save and format queries, and view query history.
- Create up to ten queries using different query tabs in the query editor. To open a new tab, click the plus sign.
- Display query results, save, and export them.
- Access the AWS Glue Data Catalog.
- View and change settings, such as view the query result location, configure auto-complete, and encrypt query results.

In the right pane, the Query Editor displays an introductory screen that prompts you to create your first table. You can view your tables under Tables in the left pane.

Here’s a high-level overview of the actions available for each table:

- **Preview tables** – View the query syntax in the Query Editor on the right.
- **Show properties** – Show a table’s name, its location in Amazon S3, input and output formats, the serialization (SerDe) library used, and whether the table has encrypted data.
- **Delete table** – Delete a table.
- **Generate CREATE TABLE DDL** – Generate the query behind a table and view it in the query editor.

Using the API

Amazon Athena enables application programming for Athena. For more information, see Amazon Athena API Reference. The latest AWS SDKs include support for the Athena API.

For examples of using the AWS SDK for Java with Athena, see Code Samples (p. 788).

For more information about AWS SDK for Java documentation and downloads, see the SDKs section in Tools for Amazon Web Services.
Using the CLI

You can access Amazon Athena using the AWS CLI. For more information, see the AWS CLI reference for Athena.
Connecting to data sources

You can use Amazon Athena to query data stored in different locations and formats in a dataset. This dataset might be in CSV, JSON, Avro, Parquet, or some other format.

The tables and databases that you work with in Athena to run queries are based on metadata. Metadata is data about the underlying data in your dataset. How that metadata describes your dataset is called the schema. For example, a table name, the column names in the table, and the data type of each column are schema, saved as metadata, that describe an underlying dataset. In Athena, we call a system for organizing metadata a data catalog or a metastore. The combination of a dataset and the data catalog that describes it is called a data source.

The relationship of metadata to an underlying dataset depends on the type of data source that you work with. Relational data sources like MySQL, PostgreSQL, and SQL Server tightly integrate the metadata with the dataset. In these systems, the metadata is most often written when the data is written. Other data sources, like those built using Hive, allow you to define metadata on-the-fly when you read the dataset. The dataset can be in a variety of formats—for example, CSV, JSON, Parquet, or Avro.

Athena natively supports the AWS Glue Data Catalog. The AWS Glue Data Catalog is a data catalog built on top of other datasets and data sources such as Amazon S3, Amazon Redshift, and Amazon DynamoDB. You can also connect Athena to other data sources by using a variety of connectors.

Topics

• Integration with AWS Glue (p. 21)
• Using Athena Data Connector for External Hive Metastore (p. 35)
• Using Amazon Athena Federated Query (p. 60)
• IAM policies for accessing data catalogs (p. 178)
• Managing data sources (p. 182)
• Connecting to Amazon Athena with ODBC and JDBC drivers (p. 183)

Integration with AWS Glue

AWS Glue is a fully managed ETL (extract, transform, and load) AWS service. One of its key abilities is to analyze and categorize data. You can use AWS Glue crawlers to automatically infer database and table schema from your data in Amazon S3 and store the associated metadata in the AWS Glue Data Catalog. Athena uses the AWS Glue Data Catalog to store and retrieve table metadata for the Amazon S3 data in your Amazon Web Services account. The table metadata lets the Athena query engine know how to find, read, and process the data that you want to query.

To create database and table schema in the AWS Glue Data Catalog, you can run an AWS Glue crawler from within Athena on a data source, or you can run Data Definition Language (DDL) queries directly in the Athena Query Editor. Then, using the database and table schema that you created, you can use Data Manipulation (DML) queries in Athena to query the data.

You can register an AWS Glue Data Catalog from an account other than your own. After you configure the required IAM permissions for AWS Glue, you can use Athena to run cross-account queries. For more information, see Cross-account access to AWS Glue data catalogs (p. 548).
For more information about the AWS Glue Data Catalog, see Populating the AWS Glue Data Catalog in the AWS Glue Developer Guide.

Separate charges apply to AWS Glue. For more information, see AWS Glue pricing and Are there separate charges for AWS Glue? (p. 34) For more information about the benefits of using AWS Glue with Athena, see Why should I upgrade to the AWS Glue Data Catalog? (p. 33)

Topics

• Using AWS Glue to connect to data sources in Amazon S3 (p. 22)
• Registering an AWS Glue Data Catalog from another account (p. 24)
• Best practices when using Athena with AWS Glue (p. 25)
• Upgrading to the AWS Glue Data Catalog step-by-step (p. 31)
• FAQ: Upgrading to the AWS Glue Data Catalog (p. 33)

Using AWS Glue to connect to data sources in Amazon S3

Athena can connect to your data stored in Amazon S3 using the AWS Glue Data Catalog to store metadata such as table and column names. After the connection is made, your databases, tables, and views appear in Athena's query editor.

To define schema information for AWS Glue to use, you can create an AWS Glue crawler to retrieve the information automatically, or you can manually add a table and enter the schema information.

Creating an AWS Glue crawler

You can create a crawler by starting in the Athena console and then using the AWS Glue console in an integrated way. When you create the crawler, you specify a data location in Amazon S3 to crawl.

To create a crawler in AWS Glue starting from the Athena console

2. In the query editor, next to Tables and views, choose Create, and then choose AWS Glue crawler.
3. On the AWS Glue console Add crawler page, follow the steps to create a crawler. For more information, see Using AWS Glue Crawlers (p. 26) in this guide and Populating the AWS Glue Data Catalog in the AWS Glue Developer Guide.

Note

Athena does not recognize exclude patterns that you specify for an AWS Glue crawler. For example, if you have an Amazon S3 bucket that contains both .csv and .json files and you exclude the .json files from the crawler, Athena queries both groups of files. To avoid this, place the files that you want to exclude in a different location.

Adding a table using a form

The following procedure shows you how to use the Athena console to add a table using the Create Table From S3 bucket data form.

To add a table and enter schema information using a form

2. In the query editor, next to Tables and views, choose Create, and then choose S3 bucket data.
3. On the Create Table From S3 bucket data form, for Table name, enter a name for the table.
4. For **Database configuration**, choose an existing database, or create a new one.

5. For **Location of Input Data Set**, specify the path in Amazon S3 to the folder that contains the dataset that you want to process. Do not include a file name in the path. Athena scans all files in the folder that you specify. If your data is already partitioned (for example, `s3://DOC-EXAMPLE-BUCKET/logs/year=2004/month=12/day=11/`), enter the base path only (for example, `s3://DOC-EXAMPLE-BUCKET/logs/`).

6. For **Data Format**, choose among the following options:
   - For **Table type**, choose **Hive**, **Governed**, or **Iceberg**. Athena uses the Apache Hive table type as the default. For information about using AWS Lake Formation governed tables in Athena, see **Using governed tables** (p. 498). For information about using Apache Iceberg tables in Athena, see **Using Iceberg tables** (p. 503).
   - For **File format**, choose the file or log format that your data is in.
   - For the **Text File with Custom Delimiters** option, specify a **Field terminator** (that is, a column delimiter). Optionally, you can specify a **Collection terminator** that marks the end of an array type or a **Collection terminator** that marks the end of a map data type.
   - **SerDe library** – A SerDe (serializer-deserializer) library parses a particular data format so that Athena can create a table for it. For most formats, a default SerDe library is chosen for you. For the following formats, choose a library according to your requirements:
     - **Apache Web Logs** – Choose either the **RegexSerDe** or **GrokSerDe** library. For RegexSerDe, provide a regular expression in the **Regex definition** box. For GrokSerDe, provide a series of named regular expressions for the `input.format` SerDe property. Named regular expressions are easier to read and maintain than regular expressions. For more information, see **Querying Apache logs stored in Amazon S3** (p. 490).
     - **CSV** – Choose **LazySimpleSerDe** if your comma-separated data does not contain values enclosed in double quotes or if it uses the `java.sql.Timestamp` format. Choose **OpenCSVSerDe** if your data includes quotes or uses the UNIX numeric format for `TIMESTAMP` (for example, `1564610311`). For more information, see **LazySimpleSerDe for CSV, TSV, and custom-delimited files** (p. 324) and **OpenCSVSerDe for processing CSV** (p. 329).
     - **JSON** – Choose either the **OpenX** or **Hive** JSON SerDe library. Both formats expect each JSON document to be on a single line of text and that fields not be separated by newline characters. The OpenX SerDe offers some additional properties. For more information about these properties, see **OpenX JSON SerDe** (p. 321). For information about the Hive SerDe, see **Hive JSON SerDe** (p. 320).

For more information about using SerDe libraries in Athena, see **Supported SerDes and data formats** (p. 303).

7. For **SerDe properties**, add, edit, or remove properties and values according to the SerDe library that you are using and your requirements.
   - To add a SerDe property, choose **Add SerDe property**.
   - In the **Name** field, enter the name of the property.
   - In the **Value** field, enter a value for the property.
   - To remove a SerDe property, choose **Remove**.

8. For **Table properties**, choose or edit the table properties according to your requirements.
   - For **Write compression**, choose a compression option. The availability of the write compression option and of the compression options available depends on the data format. For more information, see **Compression support in Athena by file format** (p. 300).
   - For **Encryption**, select **Encrypted data set** if the underlying data is encrypted in Amazon S3. This option sets the `has_encrypted_data` table property to true in the `CREATE TABLE` statement.

9. For **Column details**, enter the names and data types of the columns that you want to add to the table.
• To add more columns one at a time, choose **Add a column**.
• To quickly add more columns, choose **Bulk add columns**. In the text box, enter a comma separated list of columns in the format `column_name data_type, column_name data_type, ...`, and then choose **Add**.

10. (Optional) For **Partition details**, add one or more column names and data types. Partitioning keeps related data together based on column values and can help reduce the amount of data scanned per query. For information about partitioning, see **Partitioning data in Athena** (p. 260).

11. (Optional) For **Bucketing**, you can specify one or more columns that have rows that you want to group together, and then put those rows into multiple buckets. This allows you to query only the bucket that you want to read when the bucketed columns value is specified.

   • For **Buckets**, select one or more columns that have a large number of unique values (for example, a primary key) and that are frequently used to filter the data in your queries.
   • For **Number of buckets**, enter a number that permits files to be of optimal size. For more information, see **Top 10 Performance Tuning Tips for Amazon Athena** in the AWS Big Data Blog.
   • To specify your bucketed columns, the CREATE TABLE statement will use the following syntax:

   ```sql
   CLUSTERED BY (bucketed_columns) INTO number_of_buckets BUCKETS
   ```

   **Note**
   The **Bucketing** option is not available for the **Governed** or **Iceberg** table types.

12. The **Preview table query** box shows the CREATE TABLE statement generated by the information that you entered into the form. The preview statement cannot be edited directly. To change the statement, modify the form fields above the preview, or **create the statement directly** (p. 254) in the query editor instead of using the form.

13. Choose **Create table** to run the generated statement in the query editor and create the table.

### Registering an AWS Glue Data Catalog from another account

You can use Athena's cross-account AWS Glue catalog feature to register an AWS Glue catalog from an account other than your own. After you configure the required IAM permissions for AWS Glue and register the catalog as an Athena DataCatalog resource, you can use Athena to run cross-account queries. For information about configuring the required permissions, see **Cross-account access to AWS Glue data catalogs** (p. 548).

The following procedure shows you how to use the Athena console to configure an AWS Glue Data Catalog in an Amazon Web Services account other than your own as a data source.

**To register an AWS Glue Data Catalog from another account**

1. Follow the steps in **Cross-account access to AWS Glue data catalogs** (p. 548) to ensure that you have permissions to query the data catalog in the other account.
2. Open the Athena console at https://console.aws.amazon.com/athena/.
3. If the console navigation pane is not visible, choose the expansion menu on the left.
4. Choose **Data sources**.
5. On the upper right, choose **Create data source**.
6. On the **Choose a data source** page, for **Data sources**, choose **S3 - AWS Glue Data Catalog**, and then choose **Next**.
7. On the **Enter data source details** page, in the **AWS Glue Data Catalog** section, for **Choose an AWS Glue Data Catalog**, choose **AWS Glue Data Catalog in another account**.
8. For **Data source details**, enter the following information:
   - **Data source name** – Enter the name that you want to use in your SQL queries to refer to the data catalog in the other account.
   - **Description** – (Optional) Enter a description of the data catalog in the other account.
   - **Catalog ID** – Enter the 12-digit Amazon Web Services account ID of the account to which the data catalog belongs. The Amazon Web Services account ID is the catalog ID.
9. (Optional) For **Tags**, enter key-value pairs that you want to associate with the data source. For more information about tags, see [Tagging Athena resources](p. 658).
10. Choose **Next**.
11. On the **Review and create** page, review the information that you provided, and then choose **Create data source**. The **Data source details** page lists the databases and tags for the data catalog that you registered.
12. Choose **Data sources**. The data catalog that you registered is listed in the **Data source name** column.
13. To view or edit information about the data catalog, choose the catalog, and then choose **Actions**, **Edit**.
14. To delete the new data catalog, choose the catalog, and then choose **Actions**, **Delete**.

### Best practices when using Athena with AWS Glue

When using Athena with the AWS Glue Data Catalog, you can use AWS Glue to create databases and tables (schema) to be queried in Athena, or you can use Athena to create schema and then use them in AWS Glue and related services. This topic provides considerations and best practices when using either method.

Under the hood, Athena uses Presto to process DML statements and Hive to process the DDL statements that create and modify schema. With these technologies, there are a couple of conventions to follow so that Athena and AWS Glue work well together.

**In this topic**

- **Database, table, and column names** (p. 26)
- **Using AWS Glue crawlers** (p. 26)
  - Scheduling a crawler to keep the AWS Glue Data Catalog and Amazon S3 in sync (p. 26)
• Using multiple data sources with crawlers (p. 27)
• Syncing partition schema to avoid "HIVE_PARTITION_SCHEMA_MISMATCH" (p. 27)
• Updating table metadata (p. 28)

• Working with CSV files (p. 28)
  • CSV data enclosed in quotes (p. 28)
  • CSV files with headers (p. 29)

• AWS Glue partition indexing and filtering (p. 29)

• Working with geospatial data (p. 30)

• Using AWS Glue jobs for ETL with Athena (p. 30)
  • Creating tables using Athena for AWS Glue ETL jobs (p. 30)
  • Using ETL jobs to optimize query performance (p. 31)
  • Converting SMALLINT and TINYINT data types to INT when converting to ORC (p. 31)
  • Automating AWS Glue jobs for ETL (p. 31)

Database, table, and column names

When you create schema in AWS Glue to query in Athena, consider the following:

• A database name cannot be longer than 255 characters.
• A table name cannot be longer than 255 characters.
• A column name cannot be longer than 255 characters.
• The only acceptable characters for database names, table names, and column names are lowercase letters, numbers, and the underscore character.

For more information, see Databases and Tables in the AWS Glue Developer Guide.

Note

If you use an AWS::Glue::Database AWS CloudFormation template to create an AWS Glue database and do not specify a database name, AWS Glue automatically generates a database name in the format resource_name-random_string that is not compatible with Athena.

You can use the AWS Glue Catalog Manager to rename columns, but not table names or database names. To change a database name, you must create a new database and copy tables from the old database to it (in other words, copy the metadata to a new entity). You can follow a similar process for tables. You can use the AWS Glue SDK or AWS CLI to do this.

Using AWS Glue crawlers

AWS Glue crawlers help discover the schema for datasets and register them as tables in the AWS Glue Data Catalog. The crawlers go through your data and determine the schema. In addition, the crawler can detect and register partitions. For more information, see Defining crawlers in the AWS Glue Developer Guide. Tables from data that were successfully crawled can be queried from Athena.

Note

Athena does not recognize exclude patterns that you specify for an AWS Glue crawler. For example, if you have an Amazon S3 bucket that contains both .csv and .json files and you exclude the .json files from the crawler, Athena queries both groups of files. To avoid this, place the files that you want to exclude in a different location.

Scheduling a crawler to keep the AWS Glue Data Catalog and Amazon S3 in sync

AWS Glue crawlers can be set up to run on a schedule or on demand. For more information, see Time-based schedules for jobs and crawlers in the AWS Glue Developer Guide.
If you have data that arrives for a partitioned table at a fixed time, you can set up an AWS Glue crawler to run on schedule to detect and update table partitions. This can eliminate the need to run a potentially long and expensive MSCK REPAIR command or manually run an ALTER TABLE ADD PARTITION command. For more information, see Table partitions in the AWS Glue Developer Guide.

Using multiple data sources with crawlers

When an AWS Glue crawler scans Amazon S3 and detects multiple directories, it uses a heuristic to determine where the root for a table is in the directory structure, and which directories are partitions for the table. In some cases, where the schema detected in two or more directories is similar, the crawler may treat them as partitions instead of separate tables. One way to help the crawler discover individual tables is to add each table's root directory as a data store for the crawler.

The following partitions in Amazon S3 are an example:

```
s3://bucket01/folder1/table1/partition1/file.txt
s3://bucket01/folder1/table1/partition2/file.txt
s3://bucket01/folder1/table1/partition3/file.txt
s3://bucket01/folder1/table2/partition4/file.txt
s3://bucket01/folder1/table2/partition5/file.txt
```

If the schema for table1 and table2 are similar, and a single data source is set to s3://bucket01/folder1/ in AWS Glue, the crawler may create a single table with two partition columns: one partition column that contains table1 and table2, and a second partition column that contains partition1 through partition5.

To have the AWS Glue crawler create two separate tables, set the crawler to have two data sources, s3://bucket01/folder1/table1/ and s3://bucket01/folder1/table2, as shown in the following procedure.

**To add an S3 data store to an existing crawler in AWS Glue**

1. Sign in to the AWS Management Console and open the AWS Glue console at https://console.aws.amazon.com/glue/.
2. In the navigation pane, choose Crawlers.
3. Choose the link to your crawler, and then choose Edit.
4. For Step 2: Choose data sources and classifiers, choose Edit.
5. For Data sources, choose Add a data source.
6. In the Add data source dialog box, for S3 path, choose Browse.
7. Select the bucket that you want to use, and then choose Choose.
8. The data source that you added appears in the Data sources list.
9. Choose Next.
10. Make sure that the S3 path ends in a trailing slash, and then choose Add an S3 data source.
11. Choose Next.
12. Choose Next.
13. On the Review and update page, review the choices that you made. To edit a step, choose Edit.

**Syncing partition schema to avoid "HIVE_PARTITION_SCHEMA_MISMATCH"**

For each table within the AWS Glue Data Catalog that has partition columns, the schema is stored at the table level and for each individual partition within the table. The schema for partitions are
populated by an AWS Glue crawler based on the sample of data that it reads within the partition. For more information, see Using multiple data sources with crawlers (p. 27).

When Athena runs a query, it validates the schema of the table and the schema of any partitions necessary for the query. The validation compares the column data types in order and makes sure that they match for the columns that overlap. This prevents unexpected operations such as adding or removing columns from the middle of a table. If Athena detects that the schema of a partition differs from the schema of the table, Athena may not be able to process the query and fails with HIVE_PARTITION_SCHEMA_MISMATCH.

There are a few ways to fix this issue. First, if the data was accidentally added, you can remove the data files that cause the difference in schema, drop the partition, and re-crawl the data. Second, you can drop the individual partition and then run MSCK REPAIR within Athena to re-create the partition using the table's schema. This second option works only if you are confident that the schema applied will continue to read the data correctly.

Updating table metadata

After a crawl, the AWS Glue crawler automatically assigns certain table metadata to help make it compatible with other external technologies like Apache Hive, Presto, and Spark. Occasionally, the crawler may incorrectly assign metadata properties. Manually correct the properties in AWS Glue before querying the table using Athena. For more information, see Viewing and editing table details in the AWS Glue Developer Guide.

AWS Glue may mis-assign metadata when a CSV file has quotes around each data field, getting the serializationLib property wrong. For more information, see CSV data enclosed in quotes (p. 28).

Working with CSV files

CSV files occasionally have quotes around the data values intended for each column, and there may be header values included in CSV files, which aren't part of the data to be analyzed. When you use AWS Glue to create schema from these files, follow the guidance in this section.

CSV data enclosed in quotes

You might have a CSV file that has data fields enclosed in double quotes like the following example:

```
"John", "Doe", "123-555-1231", "John said "hello"
"Jane", "Doe", "123-555-9876", "Jane said "hello"
```

To run a query in Athena on a table created from a CSV file that has quoted values, you must modify the table properties in AWS Glue to use the OpenCSVSerDe. For more information about the OpenCSV SerDe, see OpenCSVSerDe for processing CSV (p. 329).

To edit table properties in the AWS Glue console

1. In the AWS Glue console navigation pane, choose Tables.
2. Choose the link for the table that you want to edit, and then choose Actions, Edit table.
3. On the Edit table page, make the following changes:
   - For Serialization lib, enter org.apache.hadoop.hive.serde2.OpenCSVSerde.
   - For Serde parameters, enter the following values for the keys escapeChar, quoteChar, and separatorChar:
     - For escapeChar, enter a backslash (\).
     - For quoteChar, enter a double quote (").
Best practices when using Athena with AWS Glue

4. For separatorChar, enter a comma (,).

For more information, see Viewing and editing table details in the AWS Glue Developer Guide.

Updating AWS Glue table properties programatically

You can use the AWS Glue UpdateTable API operation or update-table CLI command to modify the SerDeInfo block in the table definition, as in the following example JSON.

```
"SerDeInfo": {
    "name": "",
    "serializationLib": "org.apache.hadoop.hive.serde2.OpenCSVSerde",
    "parameters": {
        "separatorChar": ",",
        "quoteChar": "\\"
    }
},
```

CSV files with headers

When you define a table in Athena with a CREATE TABLE statement, you can use the skip.header.line.count table property to ignore headers in your CSV data, as in the following example.

```
STORED AS TEXTFILE
LOCATION 's3://my_bucket/csvdata_folder/';
TBLPROPERTIES ("skip.header.line.count"="1")
```

Alternatively, you can remove the CSV headers beforehand so that the header information is not included in Athena query results. One way to achieve this is to use AWS Glue jobs, which perform extract, transform, and load (ETL) work. You can write scripts in AWS Glue using a language that is an extension of the PySpark Python dialect. For more information, see Authoring Jobs in AWS Glue in the AWS Glue Developer Guide.

The following example shows a function in an AWS Glue script that writes out a dynamic frame using from_options, and sets the writeHeader format option to false, which removes the header information:

```
enumerate.write_dynamic_frame.from_options(frame = applymapping1, connection_type = "s3", connection_options = {"path": "s3://MYBUCKET/MYTABLEDATA/", format = "csv", format_options = {"writeHeader": False}, transformation_ctx = "datasink2")
```

AWS Glue partition indexing and filtering

When Athena queries partitioned tables, it retrieves and filters the available table partitions to the subset relevant to your query. As new data and partitions are added, more time is required to process the partitions, and query runtime can increase. If you have a table with a large number of partitions that grows over time, consider using AWS Glue partition indexing and filtering. Partition indexing allows Athena to optimize partition processing and improve query performance on highly partitioned tables. Setting up partition filtering in a table’s properties is a two-step process:

1. Creating a partition index in AWS Glue.
2. Enabling partition filtering for the table.
Creating a partition index

For steps on creating a partition index in AWS Glue, see Working with partition indexes in the AWS Glue Developer Guide. For the limitations on partition indexes in AWS Glue, see the About partition indexes section on that page.

Enabling partition filtering

To enable partition filtering for the table, you must set a new table property in AWS Glue. For steps on how to set table properties in AWS Glue, refer to the Setting up partition projection page. When you edit the table details in AWS Glue, add the following key-value pair to the Table properties section:

- For Key, add partition_filtering.enabled
- For Value, add true

You can disable partition filtering on this table at any time by setting the partition_filtering.enabled value to false.

After you complete the above steps, you can return to the Athena console to query the data.

Working with geospatial data

AWS Glue does not natively support Well-known Text (WKT), Well-Known Binary (WKB), or other PostGIS data types. The AWS Glue classifier parses geospatial data and classifies them using supported data types for the format, such as varchar for CSV. As with other AWS Glue tables, you may need to update the properties of tables created from geospatial data to allow Athena to parse these data types as-is. For more information, see Using AWS Glue crawlers (p. 26) and Working with CSV files (p. 28). Athena may not be able to parse some geospatial data types in AWS Glue tables as-is. For more information about working with geospatial data in Athena, see Querying geospatial data (p. 402).

Using AWS Glue jobs for ETL with Athena

AWS Glue jobs perform ETL operations. An AWS Glue job runs a script that extracts data from sources, transforms the data, and loads it into targets. For more information, see Authoring Jobs in AWS Glue in the AWS Glue Developer Guide.

Creating tables using Athena for AWS Glue ETL jobs

Tables that you create in Athena must have a table property added to them called a classification, which identifies the format of the data. This allows AWS Glue to use the tables for ETL jobs. The classification values can be avro, csv, json, orc, parquet, or xml. An example CREATE TABLE statement in Athena follows:

```sql
CREATE EXTERNAL TABLE sampleTable (column1 INT, column2 INT) STORED AS PARQUET TBLPROPERTIES ('classification'='parquet')
```

If the table property was not added when the table was created, you can add it using the AWS Glue console.

To add the classification table property using the AWS Glue console

1. Sign in to the AWS Management Console and open the AWS Glue console at https://console.aws.amazon.com/glue/.
2. In the console navigation pane, choose Tables.
3. Choose the link for the table that you want to edit, and then choose **Actions, Edit table**.

4. Scroll down to the **Table properties** section.

5. Choose **Add**.

6. For **Key**, enter **classification**.

7. For **Value**, enter a data type (for example, **json**).

8. Choose **Save**.

   In the **Table details** section, the data type that you entered appears in the **Classification** field for the table.

For more information, see Working with tables in the AWS Glue Developer Guide.

**Using ETL jobs to optimize query performance**

AWS Glue jobs can help you transform data to a format that optimizes query performance in Athena. Data formats have a large impact on query performance and query costs in Athena.

We recommend to use Parquet and ORC data formats. AWS Glue supports writing to both of these data formats, which can make it easier and faster for you to transform data to an optimal format for Athena. For more information about these formats and other ways to improve performance, see Top performance tuning tips for Amazon Athena.

**Converting SMALLINT and TINYINT data types to INT when converting to ORC**

To reduce the likelihood that Athena is unable to read the SMALLINT and TINYINT data types produced by an AWS Glue ETL job, convert SMALLINT and TINYINT to INT when using the wizard or writing a script for an ETL job.

**Automating AWS Glue jobs for ETL**

You can configure AWS Glue ETL jobs to run automatically based on triggers. This feature is ideal when data from outside AWS is being pushed to an Amazon S3 bucket in a suboptimal format for querying in Athena. For more information, see Triggering AWS Glue jobs in the AWS Glue Developer Guide.

**Upgrading to the AWS Glue Data Catalog step-by-step**

Currently, all regions that support Athena also support AWS Glue Data Catalog. Databases and tables are available to Athena using the AWS Glue Data Catalog and vice versa.

If you created databases and tables using Athena or Amazon Redshift Spectrum prior to a region's support for AWS Glue, you can upgrade Athena to use the AWS Glue Data Catalog.

If you are using the older Athena-managed data catalog, you see the option to upgrade at the top of the console. The metadata in the Athena-managed catalog isn't available in the AWS Glue Data Catalog or vice versa. While the catalogs exist side-by-side, creating tables or databases with the same names fails in either AWS Glue or Athena. This prevents name collisions when you do upgrade. For more information about the benefits of using the AWS Glue Data Catalog, see FAQ: Upgrading to the AWS Glue Data Catalog (p. 33).

A wizard in the Athena console can walk you through upgrading to the AWS Glue console. The upgrade takes just a few minutes, and you can pick up where you left off. For information about each upgrade step, see the topics in this section.

For information about working with data and tables in the AWS Glue Data Catalog, see the guidelines in Best practices when using Athena with AWS Glue (p. 25).
Step 1 - Allow a user to perform the upgrade

By default, the action that allows a user to perform the upgrade is not allowed in any policy, including any managed policies. Because the AWS Glue Data Catalog is shared throughout an account, this extra failsafe prevents someone from accidentally migrating the catalog.

Before the upgrade can be performed, you need to attach a customer-managed IAM policy, with a policy statement that allows the upgrade action, to the user who performs the migration.

The following is an example policy statement.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "glue:ImportCatalogToGlue"
      ],
      "Resource": [ "*" ]
    }
  ]
}
```

Step 2 - Update customer-managed or inline policies associated with Athena users

If you have customer-managed or inline IAM policies associated with Athena users, you need to update the policy or policies to allow actions that AWS Glue requires. If you use the Athena managed policy, no action is required. The AWS Glue policy actions to allow are listed in the example policy below. For the full policy statement, see AWS managed policies for Amazon Athena (p. 530).

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "glue:CreateDatabase",
        "glue:DeleteDatabase",
        "glue:GetDatabase",
        "glue:GetDatabases",
        "glue:UpdateDatabase",
        "glue:CreateTable",
        "glue:DeleteTable",
        "glue:BatchDeleteTable",
        "glue:GetTable",
        "glue:GetTables",
        "glue:BatchCreatePartition",
        "glue:CreatePartition",
        "glue:DeletePartition",
        "glue:BatchDeletePartition",
        "glue:UpdatePartition",
        "glue:GetPartition",
        "glue:GetPartitions",
        "glue:BatchGetPartition"
      ],
      "Resource": [ "*" ]
    }
  ]
}
```
Step 3 - Choose upgrade in the Athena console

After you make the required IAM policy updates, choose Upgrade in the Athena console. Athena moves your metadata to the AWS Glue Data Catalog. The upgrade takes only a few minutes.

When you create a table using the console, you can create a table using an AWS Glue crawler. For more information, see Using AWS Glue crawlers (p. 26).

FAQ: Upgrading to the AWS Glue Data Catalog

If you created databases and tables using Athena in a region before AWS Glue was available in that region, metadata is stored in an Athena-managed data catalog, which only Athena and Amazon Redshift Spectrum can access. To use AWS Glue with Athena and Redshift Spectrum, you must upgrade to the AWS Glue Data Catalog.

Why should I upgrade to the AWS Glue Data Catalog?

AWS Glue is a completely-managed extract, transform, and load (ETL) service. It has three main components:

- An AWS Glue crawler can automatically scan your data sources, identify data formats, and infer schema.
- A fully managed ETL service allows you to transform and move data to various destinations.
- The AWS Glue Data Catalog stores metadata information about databases and tables and points to a data store in Amazon S3 or a JDBC-compliant data store.

For more information, see AWS Glue concepts.

Upgrading to the AWS Glue Data Catalog has the following benefits.

Unified metadata repository

The AWS Glue Data Catalog provides a unified metadata repository across a variety of data sources and data formats. It provides out-of-the-box integration with Amazon Simple Storage Service (Amazon S3), Amazon Relational Database Service (Amazon RDS), Amazon Redshift, Amazon Redshift Spectrum, Athena, Amazon EMR, and any application compatible with the Apache Hive metastore. You can create your table definitions one time and query across engines.

For more information, see Populating the AWS Glue Data Catalog.

Automatic schema and partition recognition

AWS Glue crawlers automatically crawl your data sources, identify data formats, and suggest schema and transformations. Crawlers can help automate table creation and automatic loading of partitions that you can query using Athena, Amazon EMR, and Redshift Spectrum. You can also create tables and partitions directly using the AWS Glue API, SDKs, and the AWS CLI.

For more information, see Cataloging tables with a crawler.

Easy-to-build pipelines

The AWS Glue ETL engine generates Python code that is entirely customizable, reusable, and portable. You can edit the code using your favorite IDE or notebook and share it with others using GitHub. After
your ETL job is ready, you can schedule it to run on the fully managed, scale-out Spark infrastructure of AWS Glue. AWS Glue handles provisioning, configuration, and scaling of the resources required to run your ETL jobs, allowing you to tightly integrate ETL with your workflow.

For more information, see Authoring AWS Glue jobs in the AWS Glue Developer Guide.

Are there separate charges for AWS Glue?

Yes. With AWS Glue, you pay a monthly rate for storing and accessing the metadata stored in the AWS Glue Data Catalog, an hourly rate billed per second for AWS Glue ETL jobs and crawler runtime, and an hourly rate billed per second for each provisioned development endpoint. The AWS Glue Data Catalog allows you to store up to a million objects at no charge. If you store more than a million objects, you are charged USD$1 for each 100,000 objects over a million. An object in the AWS Glue Data Catalog is a table, a partition, or a database. For more information, see AWS Glue pricing.

Upgrade process FAQ

- Who can perform the upgrade? (p. 34)
- My users use a managed policy with Athena and Redshift Spectrum. What steps do I need to take to upgrade? (p. 34)
- What happens if I don't upgrade? (p. 34)
- Why do I need to add AWS Glue policies to Athena users? (p. 34)
- What happens if I don't allow AWS Glue policies for Athena users? (p. 35)
- Is there risk of data loss during the upgrade? (p. 35)
- Is my data also moved during this upgrade? (p. 35)

Who can perform the upgrade?

You need to attach a customer-managed IAM policy with a policy statement that allows the upgrade action to the user who will perform the migration. This extra check prevents someone from accidentally migrating the catalog for the entire account. For more information, see Step 1 - Allow a user to perform the upgrade (p. 32).

My users use a managed policy with Athena and Redshift Spectrum. What steps do I need to take to upgrade?

The Athena managed policy has been automatically updated with new policy actions that allow Athena users to access AWS Glue. However, you still must explicitly allow the upgrade action for the user who performs the upgrade. To prevent accidental upgrade, the managed policy does not allow this action.

What happens if I don't upgrade?

If you don't upgrade, you are not able to use AWS Glue features together with the databases and tables that you create in Athena or vice versa. You can use these services independently. During this time, Athena and AWS Glue both prevent you from creating databases or tables that have the same names in the other data catalog. This prevents name collisions when you do upgrade.

Why do I need to add AWS Glue policies to Athena users?

Before you upgrade, Athena manages the data catalog, so Athena actions must be allowed for your users to perform queries. After you upgrade to the AWS Glue Data Catalog, AWS Glue actions must be allowed for your users. Remember, the managed policy for Athena has already been updated to allow the required AWS Glue actions, so no action is required if you use the managed policy.
What happens if I don't allow AWS Glue policies for Athena users?

If you upgrade to the AWS Glue Data Catalog and don't update a user’s customer-managed or inline IAM policies, Athena queries fail because the user won’t be allowed to perform actions in AWS Glue. For the specific actions to allow, see Step 2 - Update customer-managed or inline policies associated with Athena users (p. 32).

Is there risk of data loss during the upgrade?

No.

Is my data also moved during this upgrade?

No. The migration only affects metadata.

Using Athena Data Connector for External Hive Metastore

You can use the Amazon Athena data connector for external Hive metastore to query data sets in Amazon S3 that use an Apache Hive metastore. No migration of metadata to the AWS Glue Data Catalog is necessary. In the Athena management console, you configure a Lambda function to communicate with the Hive metastore that is in your private VPC and then connect it to the metastore. The connection from Lambda to your Hive metastore is secured by a private Amazon VPC channel and does not use the public internet. You can provide your own Lambda function code, or you can use the default implementation of the Athena data connector for external Hive metastore.

Topics

• Overview of features (p. 35)
• Workflow (p. 36)
• Considerations and limitations (p. 37)
• Connecting Athena to an Apache Hive metastore (p. 38)
• Using the AWS Serverless Application Repository to deploy a Hive data source connector (p. 41)
• Connecting Athena to a Hive metastore using an existing IAM execution role (p. 43)
• Configure Athena to use a deployed Hive metastore connector (p. 46)
• Using a default data source name in external Hive metastore queries (p. 47)
• Working with Hive views (p. 50)
• Using the AWS CLI with Hive metastores (p. 54)
• Reference implementation (p. 59)

Overview of features

With the Athena data connector for external Hive metastore, you can perform the following tasks:

• Use the Athena console to register custom catalogs and run queries using them.
• Define Lambda functions for different external Hive metastores and join them in Athena queries.
• Use the AWS Glue Data Catalog and your external Hive metastores in the same Athena query.
• Specify a catalog in the query execution context as the current default catalog. This removes the requirement to prefix catalog names to database names in your queries. Instead of using the syntax `catalog.database.table`, you can use `database.table`. 
• Use a variety of tools to run queries that reference external Hive metastores. You can use the Athena console, the AWS CLI, the AWS SDK, Athena APIs, and updated Athena JDBC and ODBC drivers. The updated drivers have support for custom catalogs.

API support

Athena Data Connector for External Hive Metastore includes support for catalog registration API operations and metadata API operations.

• Catalog registration – Register custom catalogs for external Hive metastores and federated data sources (p. 60).
• Metadata – Use metadata APIs to provide database and table information for AWS Glue and any catalog that you register with Athena.
• Athena JAVA SDK client – Use catalog registration APIs, metadata APIs, and support for catalogs in the StartQueryExecution operation in the updated Athena Java SDK client.

Reference implementation

Athena provides a reference implementation for the Lambda function that connects to external Hive metastores. The reference implementation is provided on GitHub as an open source project at Athena Hive metastore.

The reference implementation is available as the following two AWS SAM applications in the AWS Serverless Application Repository (SAR). You can use either of these applications in the SAR to create your own Lambda functions.

• AthenaHiveMetastoreFunction – Uber Lambda function .jar file. An "uber" JAR (also known as a fat JAR or JAR with dependencies) is a .jar file that contains both a Java program and its dependencies in a single file.
• AthenaHiveMetastoreFunctionWithLayer – Lambda layer and thin Lambda function .jar file.

Workflow

The following diagram shows how Athena interacts with your external Hive metastore.
In this workflow, your database-connected Hive metastore is inside your VPC. You use Hive Server2 to manage your Hive metastore using the Hive CLI.

The workflow for using external Hive metastores from Athena includes the following steps.

1. You create a Lambda function that connects Athena to the Hive metastore that is inside your VPC.
2. You register a unique catalog name for your Hive metastore and a corresponding function name in your account.
3. When you run an Athena DML or DDL query that uses the catalog name, the Athena query engine calls the Lambda function name that you associated with the catalog name.
4. Using AWS PrivateLink, the Lambda function communicates with the external Hive metastore in your VPC and receives responses to metadata requests. Athena uses the metadata from your external Hive metastore just like it uses the metadata from the default AWS Glue Data Catalog.

Considerations and limitations

When you use Athena Data Connector for External Hive Metastore, consider the following points:

- DDL support for external Hive metastore is limited to the following statements.
  - DESCRIBE TABLE
  - SHOW COLUMNS
  - SHOW TABLES
  - SHOW SCHEMAS
  - SHOW CREATE TABLE
  - SHOW TBLPROPERTIES
  - SHOW PARTITIONS
- The maximum number of registered catalogs that you can have is 1,000.
- You can use CTAS (p. 281) to create an AWS Glue table from a query on an external Hive metastore, but not to create a table on an external Hive metastore.
• You can use INSERT INTO to insert data into an AWS Glue table from a query on an external Hive metastore, but not to insert data into an external Hive metastore.

• Kerberos authentication for Hive metastore is not supported.

• To use the JDBC driver with an external Hive metastore or federated queries (p. 60), include MetadataRetrievalMethod=ProxyAPI in your JDBC connection string. For information about the JDBC driver, see Using Athena with the JDBC driver (p. 183).

• The Hive hidden columns $path, $bucket, $file_size, $file_modified_time, $partition, $row_id cannot be used for fine-grained access control filtering.

• Hive hidden system tables like example_table$partitions or example_table$properties are not supported by fine-grained access control.

Permissions

Prebuilt and custom data connectors might require access to the following resources to function correctly. Check the information for the connector that you use to make sure that you have configured your VPC correctly. For information about required IAM permissions to run queries and create a data source connector in Athena, see Allow access to an Athena Data Connector for External Hive Metastore (p. 557) and Allow Lambda function access to external Hive metastores (p. 559).

• Amazon S3 – In addition to writing query results to the Athena query results location in Amazon S3, data connectors also write to a spill bucket in Amazon S3. Connectivity and permissions to this Amazon S3 location are required. For more information, see Spill location in Amazon S3 (p. 38) later in this topic.

• Athena – Access is required to check query status and prevent overscan.

• AWS Glue – Access is required if your connector uses AWS Glue for supplemental or primary metadata.

• AWS Key Management Service

• Policies – Hive metastore, Athena Query Federation, and UDFs require policies in addition to the AWS managed policy: AmazonAthenaFullAccess (p. 531). For more information, see Identity and access management in Athena (p. 530).

Spill location in Amazon S3

Because of the limit on Lambda function response sizes, responses larger than the threshold spill into an Amazon S3 location that you specify when you create your Lambda function. Athena reads these responses from Amazon S3 directly.

Note

Athena does not remove the response files on Amazon S3. We recommend that you set up a retention policy to delete response files automatically.

Connecting Athena to an Apache Hive metastore

To connect Athena to an Apache Hive metastore, you must create and configure a Lambda function. For a basic implementation, you can perform all required steps starting from the Athena management console.

Note

The following procedure requires that you have permission to create a custom IAM role for the Lambda function. If you do not have permission to create a custom role, you can use the Athena reference implementation (p. 36) to create a Lambda function separately, and then use the AWS Lambda console to choose an existing IAM role for the function. For more information, see Connecting Athena to a Hive metastore using an existing IAM execution role (p. 43).
To connect Athena to a Hive metastore

2. If the console navigation pane is not visible, choose the expansion menu on the left.
3. Choose Data sources.
4. On the upper right of the console, choose Create data source.
5. On the Choose a data source page, for Data sources, choose S3 - Apache Hive metastore.
6. Choose Next.
7. In the Data source details section, for Data source name, enter the name that you want to use in your SQL statements when you query the data source from Athena. The name can be up to 127 characters and must be unique within your account. It cannot be changed after you create it. Valid characters are a-z, A-Z, 0-9, _ (underscore), @ (at sign) and - (hyphen). The names awsdatacatalog, hive, jmx, and system are reserved by Athena and cannot be used for data source names.
8. For Lambda function, choose Create Lambda function, and then choose Create a new Lambda function in AWS Lambda

The AthenaHiveMetastoreFunction page opens in the AWS Lambda console. The page includes detailed information about the connector.
9. Under Application settings, enter the parameters for your Lambda function.

- **LambdaFuncName** – Provide a name for the function. For example, `myHiveMetastore`.
- **SpillLocation** – Specify an Amazon S3 location in this account to hold spillover metadata if the Lambda function response size exceeds 4MB.
- **HMSUris** – Enter the URI of your Hive metastore host that uses the Thrift protocol at port 9083. Use the syntax `thrift://<host_name>:9083`.
- **LambdaMemory** – Specify a value from 128MB to 3008MB. The Lambda function is allocated CPU cycles proportional to the amount of memory that you configure. The default is 1024.
- **LambdaTimeout** – Specify the maximum permissible Lambda invocation run time in seconds from 1 to 900 (900 seconds is 15 minutes). The default is 300 seconds (5 minutes).
- **VPCSecurityGroupIds** – Enter a comma-separated list of VPC security group IDs for the Hive metastore.
- **VPCSubnetIds** – Enter a comma-separated list of VPC subnet IDs for the Hive metastore.

10. Select **I acknowledge that this app creates custom IAM roles**, and then choose **Deploy**.
When the deployment completes, your function appears in your list of Lambda applications. Now that the Hive metastore function has been deployed to your account, you can configure Athena to use it.

11. Return to the **Enter data source details** page of the Athena console.
12. In the **Lambda function** section, choose the refresh icon next to the Lambda function search box. Refreshing the list of available functions causes your newly created function to appear in the list.
13. Choose the name of the function that you just created in the Lambda console. The ARN of the Lambda function displays.
14. (Optional) For **Tags**, add key-value pairs to associate with this data source. For more information about tags, see **Tagging Athena resources** (p. 658).
15. Choose **Next**.
16. On the **Review and create** page, review the data source details, and then choose **Create data source**.
17. The **Data source details** section of the page for your data source shows information about your new connector.

You can now use the **Data source name** that you specified to reference the Hive metastore in your SQL queries in Athena. In your SQL queries, use the following example syntax, replacing `hms-catalog-1` with the catalog name that you specified earlier.

```
SELECT * FROM hms-catalog-1.CustomerData.customers
```

18. For information about viewing, editing, or deleting the data sources that you create, see **Managing data sources** (p. 182).

**Using the AWS Serverless Application Repository to deploy a Hive data source connector**

To deploy an Athena data source connector for Hive, you can use the **AWS Serverless Application Repository** instead of starting with the Athena console. Use the AWS Serverless Application Repository to find the connector that you want to use, provide the parameters that the connector requires, and then deploy the connector to your account. Then, after you deploy the connector, you use the Athena console to make the data source available to Athena.

**To use the AWS Serverless Application Repository to deploy a data source connector for Hive to your account**

1. Sign in to the AWS Management Console and open the **Serverless App Repository**.
2. In the navigation pane, choose **Available applications**.
3. Select the option **Show apps that create custom IAM roles or resource policies**.
4. In the search box, enter **Hive**. The connectors that appear include the following two:
• **AthenaHiveMetastoreFunction** – Uber Lambda function .jar file.
• **AthenaHiveMetastoreFunctionWithLayer** – Lambda layer and thin Lambda function .jar file.

The two applications have the same functionality and differ only in their implementation. You can use either one to create a Lambda function that connects Athena to your Hive metastore.

5. Choose the name of the connector that you want to use. This tutorial uses **AthenaHiveMetastoreFunction**.

6. Under **Application settings**, enter the parameters for your Lambda function.
   - **LambdaFuncName** – Provide a name for the function. For example, **myHiveMetastore**.
   - **SpillLocation** – Specify an Amazon S3 location in this account to hold spillover metadata if the Lambda function response size exceeds 4MB.
   - **HMSUris** – Enter the URI of your Hive metastore host that uses the Thrift protocol at port 9083. Use the syntax `thrift://<host_name>:9083`.
   - **LambdaMemory** – Specify a value from 128MB to 3008MB. The Lambda function is allocated CPU cycles proportional to the amount of memory that you configure. The default is 1024.
   - **LambdaTimeout** – Specify the maximum permissible Lambda invocation run time in seconds from 1 to 900 (900 seconds is 15 minutes). The default is 300 seconds (5 minutes).
   - **VPCSecurityGroupIds** – Enter a comma-separated list of VPC security group IDs for the Hive metastore.
   - **VPCSubnetIds** – Enter a comma-separated list of VPC subnet IDs for the Hive metastore.

7. On the bottom right of the **Application details** page, select **I acknowledge that this app creates custom IAM roles**, and then choose **Deploy**.
At this point, you can configure Athena to use your Lambda function to connect to your Hive metastore. For steps, see Configure Athena to use a deployed Hive metastore connector (p. 46).

**Connecting Athena to a Hive metastore using an existing IAM execution role**

To connect your external Hive metastore to Athena with a Lambda function that uses an existing IAM role, you can use Athena’s reference implementation of the Athena connector for external Hive metastore.

The three major steps are as follows:

1. **Clone and build (p. 43)** – Clone the Athena reference implementation and build the JAR file that contains the Lambda function code.
2. **AWS Lambda console (p. 44)** – In the AWS Lambda console, create a Lambda function, assign it an existing IAM execution role, and upload the function code that you generated.
3. **Amazon Athena console (p. 46)** – In the Amazon Athena console, create a data source name that you can use to refer to your external Hive metastore in your Athena queries.

If you already have permissions to create a custom IAM role, you can use a simpler workflow that uses the Athena console and the AWS Serverless Application Repository to create and configure a Lambda function. For more information, see Connecting Athena to an Apache Hive metastore (p. 38).

**Prerequisites**

- Git must be installed on your system.
- You must have Apache Maven installed.
- You have an IAM execution role that you can assign to the Lambda function. For more information, see Allow Lambda function access to external Hive metastores (p. 559).

**Clone and build the Lambda function**

The function code for the Athena reference implementation is a Maven project located on GitHub at awslabs/aws-athena-hive-metastore. For detailed information about the project, see the corresponding README file on GitHub or the Reference implementation (p. 59) topic in this documentation.

**To clone and build the Lambda function code**

1. Enter the following command to clone the Athena reference implementation:

   ```bash
git clone https://github.com/awslabs/aws-athena-hive-metastore
   ``

2. Run the following command to build the `.jar` file for the Lambda function:

   ```bash
mvn clean install
   ``

   After the project builds successfully, the following `.jar` file is created in the target folder of your project:

   `hms-lambda-func-1.0-SNAPSHOT-withdep.jar`

   In the next section, you use the AWS Lambda console to upload this file to your Amazon Web Services account.
Create and configure the Lambda function in the AWS Lambda console

In this section, you use the AWS Lambda console to create a function that uses an existing IAM execution role. After you configure a VPC for the function, you upload the function code and configure the environment variables for the function.

Create the Lambda function

In this step, you create a function in the AWS Lambda console that uses an existing IAM role.

To create a Lambda function that uses an existing IAM role

1. Sign in to the AWS Management Console and open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. In the navigation pane, choose Functions.
3. Choose Create function.
4. Choose Author from scratch.
5. For Function name, enter the name of your Lambda function (for example, EHMSBasedLambda).
6. For Runtime, choose Java 8.
7. Under Permissions, expand Change default execution role.
8. For Execution role, choose Use an existing role.
9. For Existing role, choose the IAM execution role that your Lambda function will use for Athena (this example uses a role called AthenaLambdaExecutionRole).
11. Select Enable Network.
12. For VPC, choose the VPC that your function will have access to.
13. For Subnets, choose the VPC subnets for Lambda to use.
14. For Security groups, choose the VPC security groups for Lambda to use.
15. Choose Create function. The AWS Lambda console opens the configuration page for your function and begins creating your function.

Upload the code and configure the Lambda function

When the console informs you that your function has been successfully created, you are ready to upload the function code and configure its environment variables.

To upload your Lambda function code and configure its environment variables

1. In the Lambda function code, make sure that you are on the Code tab of the page of the function that you specified.
2. For Code source, choose Upload from, and then choose .zip or .jar file.
3. Upload the hms-lambda-func-1.0-SNAPSHOT-withdep.jar file that you generated previously.
4. On your Lambda function page, choose the Configuration tab.
5. From the pane on the left, choose Environment variables.
6. In the Environment variables section, choose Edit.
7. On the **Edit environment variables** page, use the **Add environment variable** option to add the following environment variable keys and values:

- **HMS URIS** – Use the following syntax to enter the URI of your Hive metastore host that uses the Thrift protocol at port 9083.

  \[
  \text{thrift://<host_name>:9083}
  \]

- **SPILL_LOCATION** – Specify an Amazon S3 location in your Amazon Web Services account to hold spillover metadata if the Lambda function response size exceeds 4MB.
After you have deployed a Lambda data source connector like AthenaHiveMetastoreFunction to your account, you can configure Athena to use it. To do so, you create a data source name that refers to your external Hive metastore to use in your Athena queries.

**To connect Athena to your Hive metastore using an existing Lambda function**

2. If the console navigation pane is not visible, choose the expansion menu on the left.
3. Choose **Data sources**.
4. On the **Data sources** page, choose **Create data source**.
5. On the **Choose a data source** page, for **Data sources**, choose **S3 - Apache Hive metastore**.
6. Choose **Next**.
7. In the **Data source details** section, for **Data source name**, enter the name that you want to use in your SQL statements when you query the data source from Athena (for example, MyHiveMetastore). The name can be up to 127 characters and must be unique within your account. It cannot be changed after you create it. Valid characters are a-z, A-Z, 0-9, _ (underscore), @ (at sign) and - (hyphen). The names awsdatacatalog, hive, jmx, and system are reserved by Athena and cannot be used for data source names.
8. In the **Connection details** section, use the **Select or enter a Lambda function** box to choose the name of the function that you just created. The ARN of the Lambda function displays.
9. (Optional) For **Tags**, add key-value pairs to associate with this data source. For more information about tags, see **Tagging Athena resources** (p. 658).
10. Choose **Next**.
11. On the **Review and create** page, review the data source details, and then choose **Create data source**.
12. The **Data source details** section of the page for your data source shows information about your new connector.

You can now use the **Data source name** that you specified to reference the Hive metastore in your SQL queries in Athena.

In your SQL queries, use the following example syntax, replacing ehms-catalog with the data source name that you specified earlier.

```
SELECT * FROM ehms-catalog.CustomerData.customers
```

13. To view, edit, or delete the data sources that you create, see **Managing data sources** (p. 182).

### Using a default data source name in external Hive metastore queries

When you run DML and DDL queries on external Hive metastores, you can simplify your query syntax by omitting the catalog name if that name is selected in the query editor. Certain restrictions apply to this functionality.
DML statements

To run queries with registered catalogs

1. You can put the data source name before the database using the syntax 
   `[[data_source_name].database_name].table_name`, as in the following example.
   ```
   select * from "hms-catalog-1".hms_tpch.customer limit 10;
   ```

2. When the data source that you want to use is already selected in the query editor, you can omit the
   name from the query, as in the following example.
   ```
   select * from hms_tpch.customer limit 10:
   ```

3. When you use multiple data sources in a query, you can omit only the default data source name, and
   must specify the full name for any non-default data sources.

   For example, suppose AwsDataCatalog is selected as the default data source in the query editor. The
   FROM statement in the following query excerpt fully qualifies the first two data source names but omits
   the name for the third data source because it is in the AWS Glue data catalog.
   ```
   FROM ehms01.hms_tpch.customer,
       "hms-catalog-1".hms_tpch.orders,
       hms_tpch.lineitem
   ```

DDL statements

The following Athena DDL statements support catalog name prefixes. Catalog name prefixes in other
DDL statements cause syntax errors.

```sql
SHOW TABLES [IN [catalog_name.]database_name] ["regular_expression"]
SHOW TBLPROPERTIES [[catalog_name.]database_name.]table_name [("property_name")]
SHOW COLUMNS IN [[catalog_name.]database_name.]table_name
```
SHOW PARTITIONS [[catalog_name.]database_name.]table_name
SHOW CREATE TABLE [[catalog_name.][database_name.]table_name
DESCRIBE [EXTENDED | FORMATTED] [[catalog_name.] [database_name.]table_name [PARTITION partition_spec] [col_name ( [.field_name] | [.elem$] | [.key$] | [.value$] )]]

As with DML statements, you can omit the datasource and database prefixes from the query when the data source and database are selected in the query editor.

In the following image, the hms-catalog-1 data source and the hms_tpch database are selected in the query editor. The show create table customer statement succeeds even though the hms-catalog-1 prefix and the hms_tpch database name are omitted from the query itself.

Specifying a default data source in a JDBC connection string

When you use the Athena JDBC Driver to connect Athena to an external Hive metastore, you can use the Catalog parameter to specify the default data source name in your connection string in a SQL editor like SQL workbench.

Note  
To download the latest Athena JDBC drivers, see Using Athena with the JDBC driver.

The following connection string specifies the default data source hms-catalog-name.

```
jdbc:awsathena://AwsRegion=us-east-1;S3OutputLocation=s3://<location>/lambda/results;/Workgroup=AmazonAthenaPreviewFunctionality;Catalog=hms-catalog-name;
```

The following image shows a sample JDBC connection URL as configured in SQL Workbench.
Working with Hive views

You can use Athena to query existing views in your external Apache Hive metastores. Athena translates your views for you on-the-fly at runtime without changing the original view or storing the translation.

For example, suppose you have a Hive view like the following that uses a syntax not supported in Athena like LATERAL VIEW explode():

```
CREATE VIEW team_view AS
SELECT team, score
FROM matches
LATERAL VIEW explode(scores) m AS score
```

Athena translates the Hive view query string into a statement like the following that Athena can run:

```
SELECT team, score
FROM matches
CROSS JOIN UNNEST(scores) AS m (score)
```

For information about connecting an external Hive metastore to Athena, see Using Athena Data Connector for External Hive Metastore (p. 35).
Considerations and limitations

When querying Hive views from Athena, consider the following points:

- Athena does not support creating Hive views. You can create Hive views in your external Hive metastore, which you can then query from Athena.
- Athena does not support custom UDFs for Hive views.
- Due to a known issue in the Athena console, Hive views appear under the list of tables instead of the list of views.
- Although the translation process is automatic, certain Hive functions are not supported for Hive views or require special handling. For more information, see the following section.

Hive function support limitations

This section highlights the Hive functions that Athena does not support for Hive views or that require special treatment. Currently, because Athena primarily supports functions from Hive 2.2.0, functions that are available only in higher versions (such as Hive 4.0.0) are not available. For a full list of Hive functions, see Hive language manual UDF.

Aggregate functions

Aggregate functions that require special handling

The following aggregate function for Hive views requires special handling.

- `Avg` – Instead of `avg(INT i)`, use `avg(CAST(i AS DOUBLE))`.

Aggregate functions not supported

The following Hive aggregate functions are not supported in Athena for Hive views.

```
covar_pop
histogram_numeric
ntile
percentile
percentile_approx
```

Regression functions like `regr_count`, `regr_r2`, and `regr_sxx` are not supported in Athena for Hive views.

Date functions not supported

The following Hive date functions are not supported in Athena for Hive views.

```
date_format(date/timestamp/string ts, string fmt)
day(string date)
dayofmonth(date)
extract(field FROM source)
hour(string date)
minute(string date)
month(string date)
quarter(date/timestamp/string)
second(string date)
weekofyear(string date)
year(string date)
```
Masking functions not supported

Hive masking functions like `mask()`, and `mask_first_n()` are not supported in Athena for Hive views.

Miscellaneous functions

Miscellaneous functions that require special handling

The following miscellaneous functions for Hive views require special handling.

- **md5** – Athena supports `md5(binary)` but not `md5(varchar)`.
- **Explode** – Athena supports `explode` when it is used in the following syntax:
  
  ```sql
  LATERAL VIEW [OUTER] EXPLODE(<argument>)
  ```

- **Posexplode** – Athena supports `posexplode` when it is used in the following syntax:
  
  ```sql
  LATERAL VIEW [OUTER] POSEXPLODE(<argument>)
  ```

  In the `(pos, val)` output, Athena treats the `pos` column as `BIGINT`. Because of this, you may need to cast the `pos` column to `BIGINT` to avoid a stale view. The following example illustrates this technique.

  ```sql
  SELECT CAST(c AS BIGINT) AS c_bigint, d 
  FROM table LATERAL VIEW POSEXPLODE(<argument>) t AS c, d
  ```

Miscellaneous functions not supported

The following Hive functions are not supported in Athena for Hive views.

```
aes_decrypt
aes_encrypt
current_database
current_user
inline
java_method
logged_in_user
reflect
sha/sha1/sha2
stack
version
```

Operators

Operators that require special handling

The following operators for Hive views require special handling.

- **Mod operator (%)** – Because the `DOUBLE` type implicitly casts to `DECIMAL(x,y)`, the following syntax can cause a View is stale error message:

  ```sql
  a_double % 1.0 AS column
  ```

  To work around this issue, use `CAST`, as in the following example.

  ```sql
  CAST(a_double % 1.0 as DOUBLE) AS column
  ```
• **Division operator (/)** – In Hive, int divided by int produces a double. In Athena, the same operation produces a truncated int.

**Operators not supported**

Athena does not support the following operators for Hive views.

~A – bitwise NOT  
A ^ b – bitwise XOR  
A & b – bitwise AND  
A | b – bitwise OR  
A <=> b – Returns same result as the equals (=) operator for non-null operands. Returns TRUE if both are NULL, FALSE if one of them is NULL.

**String functions**

**String functions that require special handling**

The following Hive string functions for Hive views require special handling.

• `chr(bigint|double a)` – Hive allows negative arguments; Athena does not.  
• `instr(string str, string substr)` – Because the Athena mapping for the instr function returns BIGINT instead of INT, use the following syntax:

```sql
CAST(instr(string str, string substr) as INT)
```

Without this step, the view will be considered stale.

• `length(string a)` – Because the Athena mapping for the length function returns BIGINT instead of INT, use the following syntax so that the view will not be considered stale:

```sql
CAST(length(string str) as INT)
```

**String functions not supported**

The following Hive string functions are not supported in Athena for Hive views.

```sql
ascii(string str)  
character_length(string str)  
decode(binary bin, string charset)  
encode(string src, string charset)  
elt(N int,str1 string,str2 string,str3 string,...)  
field(val1 T,val1 T,val2 T,val3 T,...)  
find_in_set(string str, string strList)  
initcap(string A)  
levenshtein(string A, string B)  
locate(string substr, string str[, int pos])  
ocet_length(string str)  
parse_url(string urlString, string partToExtract [, string keyToExtract])  
printf(String format, Obj... args)  
quote(String text)  
regexp_extract(string subject, string pattern, int index)  
repeat(string str, int n)  
sentences(string str, string lang, string locale)
```
soundex(string A)
space(int n)
str_to_map(text[, delimiter1, delimiter2])
substring_index(string A, string delim, int count)

**XPath functions not supported**

Hive XPath functions like xpath, xpath_short, and xpath_int are not supported in Athena for Hive views.

**Troubleshooting**

When you use Hive views in Athena, you may encounter the following issues:

- **View `<view name>` is stale** – This message usually indicates a type mismatch between the view in Hive and Athena. If the same function in the Hive Language Manual UDF and Presto functions and operators documentation has different signatures, try casting the mismatched data type.
- **Function not registered** – Athena does not currently support the function. For details, see the information earlier in this document.

**Using the AWS CLI with Hive metastores**

You can use aws athena CLI commands to manage the Hive metastore data catalogs that you use with Athena. After you have defined one or more catalogs to use with Athena, you can reference those catalogs in your aws athena DDL and DML commands.

**Using the AWS CLI to manage Hive metastore catalogs**

**Registering a catalog: Create-data-catalog**

To register a data catalog, you use the create-data-catalog command. Use the name parameter to specify the name that you want to use for the catalog. Pass the ARN of the Lambda function to the metadata-function option of the parameters argument. To create tags for the new catalog, use the tags parameter with one or more space-separated Key=Value argument pairs.

The following example registers the Hive metastore catalog named hms-catalog-1. The command has been formatted for readability.

```
$ aws athena create-data-catalog
  --name "hms-catalog-1"
  --type "HIVE"
  --description "Hive Catalog 1"
  --tags Key=MyKey,Value=MyValue
  --region us-east-1
```

**Showing catalog details: Get-data-catalog**

To show the details of a catalog, pass the name of the catalog to the get-data-catalog command, as in the following example.

```
$ aws athena get-data-catalog --name "hms-catalog-1" --region us-east-1
```

The following sample result is in JSON format.
Listing registered catalogs: List-data-catalogs

To list the registered catalogs, use the `list-data-catalogs` command and optionally specify a Region, as in the following example. The catalogs listed always include AWS Glue.

```bash
$ aws athena list-data-catalogs --region us-east-1
```

The following sample result is in JSON format.

```json
{
  "DataCatalogs": [
    {
      "CatalogName": "hms-catalog-1",
      "Type": "HIVE",
      "Parameters": {
        "sdk-version": "1.0"
      }
    }
  ]
}
```

Updating a catalog: Update-data-catalog

To update a data catalog, use the `update-data-catalog` command, as in the following example. The command has been formatted for readability.

```bash
$ aws athena update-data-catalog
   --name "hms-catalog-1"
   --type "HIVE"
   --description "My New Hive Catalog Description"
   --region us-east-1
```

Deleting a catalog: Delete-data-catalog

To delete a data catalog, use the `delete-data-catalog` command, as in the following example.

```bash
$ aws athena delete-data-catalog --name "hms-catalog-1" --region us-east-1
```
Showing database details: Get-database

To show the details of a database, pass the name of the catalog and the database to the `get-database` command, as in the following example.

```bash
$ aws athena get-database --catalog-name hms-catalog-1 --database-name mydb
```

The following sample result is in JSON format.

```json
{
   "Database": {
      "Name": "mydb",
      "Description": "My database",
      "Parameters": {
         "CreatedBy": "Athena",
         "EXTERNAL": "TRUE"
      }
   }
}
```

Listing databases in a catalog: List-databases

To list the databases in a catalog, use the `list-databases` command and optionally specify a Region, as in the following example.

```bash
$ aws athena list-databases --catalog-name AwsDataCatalog --region us-west-2
```

The following sample result is in JSON format.

```json
{
   "DatabaseList": [
      {
         "Name": "default"
      },
      {
         "Name": "mycrawlerdatabase"
      },
      {
         "Name": "mydatabase"
      },
      {
         "Name": "sampledb",
         "Description": "Sample database",
         "Parameters": {
            "CreatedBy": "Athena",
            "EXTERNAL": "TRUE"
         }
      },
      {
         "Name": "tpch100"
      }
   ]
}
```

Showing table details: Get-table-metadata

To show the metadata for a table, including column names and datatypes, pass the name of the catalog, database, and table name to the `get-table-metadata` command, as in the following example.
$ aws athena get-table-metadata --catalog-name AwsDataCatalog --database-name mydb --table-name cityuseragent

The following sample result is in JSON format.

```
{
    "TableMetadata": {
        "Name": "cityuseragent",
        "CreateTime": 1586451276.0,
        "LastAccessTime": 0.0,
        "TableType": "EXTERNAL_TABLE",
        "Columns": [
            {
                "Name": "city",
                "Type": "string"
            },
            {
                "Name": "useragent1",
                "Type": "string"
            }
        ],
        "PartitionKeys": [],
        "Parameters": {
            "COLUMN_STATS_ACCURATE": "false",
            "EXTERNAL": "true",
            "inputformat": "org.apache.hadoop.mapred.TextInputFormat",
            "last_modified_by": "hadoop",
            "last_modified_time": "1586454879",
            "location": "s3://athena-data/",
            "numFiles": "1",
            "numRows": "-1",
            "outputformat": "org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat",
            "rawDataSize": "-1",
            "serde.param.serialization.format": "1",
            "serde.serialization.lib": "org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe",
            "totalSize": "61"
        }
    }
}
```

**Showing metadata for all tables in a database: List-table-metadata**

To show the metadata for all tables in a database, pass the name of the catalog and database name to the list-table-metadata command. The list-table-metadata command is similar to the get-table-metadata command, except that you do not specify a table name. To limit the number of results, you can use the --max-results option, as in the following example.

$ aws athena list-table-metadata --catalog-name AwsDataCatalog --database-name sampledb --region us-east-1 --max-results 2

The following sample result is in JSON format.

```
[
    "TableMetadataList": [
        {
            "Name": "cityuseragent",
            "CreateTime": 1586451276.0,
            "LastAccessTime": 0.0,
            "TableType": "EXTERNAL_TABLE",
```
"Columns": [
  {
    "Name": "city",
    "Type": "string"
  },
  {
    "Name": "useragent1",
    "Type": "string"
  }
],
"PartitionKeys": [],
"Parameters": {
  "COLUMN_STATS_ACCURATE": "false",
  "EXTERNAL": "TRUE",
  "inputformat": "org.apache.hadoop.mapred.TextInputFormat",
  "last_modified_by": "hadoop",
  "last_modified_time": "1586454879",
  "location": "s3://athena-data/",
  "numFiles": "1",
  "numRows": "-1",
  "outputformat": "org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat",
  "rawDataSize": "-1",
  "serde.param.serialization.format": "1",
  "serde.serialization.lib": "org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe",
  "totalSize": "61"
},
"Clearinghouse_data",
"CreateTime": 1589255544.0,
"LastAccessTime": 0.0,
"TableType": "EXTERNAL_TABLE",
"Columns": [
  {
    "Name": "location",
    "Type": "string"
  },
  {
    "Name": "stock_count",
    "Type": "int"
  },
  {
    "Name": "quantity_shipped",
    "Type": "int"
  }
],
"PartitionKeys": [],
"Parameters": {
  "EXTERNAL": "TRUE",
  "inputformat": "org.apache.hadoop.mapred.TextInputFormat",
  "location": "s3://myjasondata/",
  "outputformat": "org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat",
  "serde.param.serialization.format": "1",
  "serde.serialization.lib": "org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe",
  "transient_lastDdlTime": "1589255544"
}
],
"NextToken": "eyJsYXN0RXZhbHVhdGVkS2V5Ijp7IkhBU0hfS0VZIjp7InMiOiJ0Ljk0YWZjYjk1MjJjNTQ1YmU4Y2I5OTg0MjFjYTkyZl0sInNlY29uZHMiOjE1ODkzNDIwMjIsIm5hbm9zIjo2NTUwMDAwMDB9fQ=="
Running DDL and DML statements

When you use the AWS CLI to run DDL and DML statements, you can pass the name of the Hive metastore catalog in one of two ways:

- Directly into the statements that support it.
- To the `--query-execution-context Catalog` parameter.

**DDL statements**

The following example passes in the catalog name directly as part of the `show create table` DDL statement. The command has been formatted for readability.

```
$ aws athena start-query-execution
   --query-string "show create table hms-catalog-1.hms_tpch_partitioned.lineitem"
   --result-configuration "OutputLocation=s3://mybucket/lambda/results"
```

The following example DDL `show create table` statement uses the `Catalog` parameter of `--query-execution-context` to pass the Hive metastore catalog name `hms-catalog-1`. The command has been formatted for readability.

```
$ aws athena start-query-execution
   --query-string "show create table lineitem"
   --query-execution-context "Catalog=hms-catalog-1,Database=hms_tpch_partitioned"
   --result-configuration "OutputLocation=s3://mybucket/lambda/results"
```

**DML statements**

The following example DML `select` statement passes the catalog name into the query directly. The command has been formatted for readability.

```
$ aws athena start-query-execution
   --query-string "select * from hms-catalog-1.hms_tpch_partitioned.customer limit 100"
   --result-configuration "OutputLocation=s3://mybucket/lambda/results"
```

The following example DML `select` statement uses the `Catalog` parameter of `--query-execution-context` to pass in the Hive metastore catalog name `hms-catalog-1`. The command has been formatted for readability.

```
$ aws athena start-query-execution
   --query-string "select * from customer limit 100"
   --query-execution-context "Catalog=hms-catalog-1,Database=hms_tpch_partitioned"
   --result-configuration "OutputLocation=s3://mybucket/lambda/results"
```

**Reference implementation**


The reference implementation is an Apache Maven project that has the following modules:

- **hms-service-api** – Contains the API operations between the Lambda function and the Athena service clients. These API operations are defined in the `HiveMetaStoreService` interface. Because this is a service contract, you should not change anything in this module.
• **hms-lambda-handler** – A set of default Lambda handlers that process all Hive metastore API calls. The class `MetadataHandler` is the dispatcher for all API calls. You do not need to change this package.

• **hms-lambda-layer** – A Maven assembly project that puts `hms-service-api`, `hms-lambda-handler`, and their dependencies into a .zip file. The .zip file is registered as a Lambda layer for use by multiple Lambda functions.

• **hms-lambda-func** – An example Lambda function that has the following components.
  
  • **HiveMetaStoreLambdaFunc** – An example Lambda function that extends `MetadataHandler`.
  
  • **ThriftHiveMetaStoreClient** – A Thrift client that communicates with Hive metastore. This client is written for Hive 2.3.0. If you use a different Hive version, you might need to update this class to ensure that the response objects are compatible.

  • **ThriftHiveMetaStoreClientFactory** – Controls the behavior of the Lambda function. For example, you can provide your own set of handler providers by overriding the `getHandlerProvider()` method.

• **hms.properties** – Configures the Lambda function. Most cases require updating the following two properties only.
  
  • `hive.metastore.uris` – the URI of the Hive metastore in the format `thrift://<host_name>:9083`.

  • `hive.metastore.response.spill.location` - The Amazon S3 location to store response objects when their sizes exceed a given threshold (for example, 4MB). The threshold is defined in the property `hive.metastore.response.spill.threshold`. Changing the default value is not recommended.

  **Note**

  These two properties can be overridden by the Lambda environment variables `HMS_URIS` and `SPILL_LOCATION`. Use these variables instead of recompiling the source code for the Lambda function when you want to use the function with a different Hive metastore or spill location.

### Building the artifacts yourself

Most use cases do not require you to modify the reference implementation. However, if necessary, you can modify the source code, build the artifacts yourself, and upload them to an Amazon S3 location.

Before you build the artifacts, update the properties `hive.metastore.uris` and `hive.metastore.response.spill.location` in the `hms.properties` file in the `hms-lambda-func` module.

To build the artifacts, you must have Apache Maven installed and run the command `mvn install`. This generates the layer .zip file in the output folder called `target` in the module `hms-lambda-layer` and the Lambda function .jar file in the module `hms-lambda-func`.

### Using Amazon Athena Federated Query

If you have data in sources other than Amazon S3, you can use Athena Federated Query to query the data in place or build pipelines that extract data from multiple data sources and store them in Amazon S3. With Athena Federated Query, you can run SQL queries across data stored in relational, non-relational, object, and custom data sources.

Athena uses **data source connectors** that run on AWS Lambda to run federated queries. A data source connector is a piece of code that can translate between your target data source and Athena. You can think of a connector as an extension of Athena's query engine. Prebuilt Athena data source connectors exist for data sources like Amazon CloudWatch Logs, Amazon DynamoDB, Amazon DocumentDB, and
Amazon RDS, and JDBC-compliant relational data sources such as MySQL, and PostgreSQL under the Apache 2.0 license. You can also use the Athena Query Federation SDK to write custom connectors. To choose, configure, and deploy a data source connector to your account, you can use the Athena and Lambda consoles or the AWS Serverless Application Repository. After you deploy data source connectors, the connector is associated with a catalog that you can specify in SQL queries. You can combine SQL statements from multiple catalogs and span multiple data sources with a single query.

When a query is submitted against a data source, Athena invokes the corresponding connector to identify parts of the tables that need to be read, manages parallelism, and pushes down filter predicates. Based on the user submitting the query, connectors can provide or restrict access to specific data elements. Connectors use Apache Arrow as the format for returning data requested in a query, which enables connectors to be implemented in languages such as C, C++, Java, Python, and Rust. Since connectors are processed in Lambda, they can be used to access data from any data source on the cloud or on-premises that is accessible from Lambda.

To write your own data source connector, you can use the Athena Query Federation SDK to customize one of the prebuilt connectors that Amazon Athena provides and maintains. You can modify a copy of the source code from the GitHub repository and then use the Connector publish tool to create your own AWS Serverless Application Repository package.

Note
Third party developers may have used the Athena Query Federation SDK to write data source connectors. For support or licensing issues with these data source connectors, please work with your connector provider. These connectors are not tested or supported by AWS.

For a list of data source connectors written and tested by Athena, see Using Athena data source connectors (p. 67).

For information about writing your own data source connector, see Example Athena connector on GitHub.

**Considerations and limitations**

- **Engine versions** – Athena Federated Query is supported only on Athena engine version 2 and later versions. For information about Athena engine versions, see Athena engine versioning (p. 669).
- **Views** – You cannot use views with federated data sources.
- **Write operations** – Write operations like INSERT INTO (p. 697) are not supported. Attempting to do so may result in the error message This operation is currently not supported for external catalogs.
- **Pricing** – For pricing information, see Amazon Athena pricing.

- **JDBC driver** – To use the JDBC driver with federated queries or an external Hive metastore (p. 35), include MetadataRetrievalMethod=ProxyAPI in your JDBC connection string. For information about the JDBC driver, see Using Athena with the JDBC driver (p. 183).
- **Secrets Manager** – To use the Athena Federated Query feature with AWS Secrets Manager, you must configure an Amazon VPC private endpoint for Secrets Manager. For more information, see Create a Secrets Manager VPC private endpoint in the AWS Secrets Manager User Guide.

Data source connectors might require access to the following resources to function correctly. If you use a prebuilt connector, check the information for the connector to ensure that you have configured your VPC correctly. Also, ensure that IAM principals running queries and creating connectors have privileges to required actions. For more information, see Example IAM permissions policies to allow Athena Federated Query (p. 562).

- **Amazon S3** – In addition to writing query results to the Athena query results location in Amazon S3, data connectors also write to a spill bucket in Amazon S3. Connectivity and permissions to this Amazon S3 location are required.
• **Athena** – Data sources need connectivity to Athena and vice versa for checking query status and preventing overscan.

• **AWS Glue Data Catalog** – Connectivity and permissions are required if your connector uses Data Catalog for supplemental or primary metadata.

For the most up-to-date information about known issues and limitations, see Limitations and issues in the aws-athena-query-federation GitHub repository.

**Videos**

Watch the following videos to learn more about using Athena Federated Query.

**Video: Analyze Results of Federated Query in Amazon Athena in Amazon QuickSight**

The following video demonstrates how to analyze results of an Athena federated query in Amazon QuickSight.

**Video: Game Analytics Pipeline**

The following video shows how to deploy a scalable serverless data pipeline to ingest, store, and analyze telemetry data from games and services using Amazon Athena federated queries.

**Deploying a connector and connecting to a data source**

Preparing to create federated queries is a two-part process: deploying a Lambda function data source connector, and connecting the Lambda function to a data source. In this process, you give the Lambda function a name that you can later choose in the Athena console and give the connector a name that you can reference in your SQL queries.

**Note**

To use the Athena Federated Query feature with AWS Secrets Manager, you must configure an Amazon VPC private endpoint for Secrets Manager. For more information, see Create a Secrets Manager VPC private endpoint in the AWS Secrets Manager User Guide.

**Part 1: Deploying a data source connector**

To choose, name, and deploy a data source connector, you use the Athena and Lambda consoles in an integrated process.

**Note**

To use the Amazon Athena Federated Query feature, set your workgroup to Athena engine version 2. For steps, see Changing Athena engine versions (p. 669).

**To deploy a data source connector**

2. If the console navigation pane is not visible, choose the expansion menu on the left.
3. In the navigation pane, choose Data sources.

4. On the Data sources page, choose Create data source.

5. For Choose a data source, choose the data source that you want Athena to query, considering the following guidelines:

- Choose a federated query option that corresponds to your data source. Athena has prebuilt data source connectors that you can configure for sources including MySQL, Amazon DocumentDB, and PostgreSQL.

- Choose S3 - AWS Glue Data Catalog if you want to query data in Amazon S3 and you are not using an Apache Hive metastore or one of the other federated query data source options on this page. Athena uses the AWS Glue Data Catalog to store metadata and schema information for data sources in Amazon S3. This is the default (non-federated) option. For more information, see Using AWS Glue to connect to data sources in Amazon S3 (p. 22).

- Choose S3 - Apache Hive metastore to query data sets in Amazon S3 that use an Apache Hive metastore. For more information about this option, see Connecting Athena to an Apache Hive metastore (p. 38).

- Choose Custom or shared connector if you want to create your own data source connector for use with Athena. For information about writing a data source connector, see Writing a data source connector using the Athena Query Federation SDK (p. 177).

This tutorial chooses Amazon CloudWatch Logs as the federated data source.

6. Choose Next.

7. On the Enter data source details page, for Data source name, enter the name that you want to use in your SQL statements when you query the data source from Athena (for example, CloudWatchLogs). The name can be up to 127 characters and must be unique within your account. It cannot be changed after you create it. Valid characters are a-z, A-Z, 0-9, _ (underscore), @ (at sign) and - (hyphen). The names awsdatacatalog, hive, jmx, and system are reserved by Athena and cannot be used for data source names.

8. For Lambda function, choose Create Lambda function. The function page for the connector that you chose opens in the AWS Lambda console. The page includes detailed information about the connector.

9. Under Application settings, read the description for each application setting carefully, and then enter values that correspond to your requirements.

The application settings that you see vary depending on the connector for your data source. The minimum required settings include:

- AthenaCatalogName – A name, in lower case, for the Lambda function that indicates the data source that it targets, such as cloudwatchlogs.

- SpillBucket – An Amazon S3 bucket in your account to store data that exceeds Lambda function response size limits.
Note
Spilled data is not reused in subsequent executions and can be safely deleted after 12 hours. Athena does not delete this data for you. To manage these objects, consider adding an object lifecycle policy that deletes old data from your Amazon S3 spill bucket. For more information, see Managing your storage lifecycle in the Amazon S3 User Guide.

10. Select I acknowledge that this app creates custom IAM roles and resource policies. For more information, choose the Info link.
11. Choose Deploy. When the deployment is complete, the Lambda function appears in the Resources section in the Lambda console.

Part 2: Connecting to the data source

After you deploy the data source connector to your account, you can connect Athena to it.

To connect Athena to a data source using a connector that you have deployed to your account

1. Return to the Enter data source details page of the Athena console.
2. In the Connection details section, choose the refresh icon next to the Select or enter a Lambda function search box.
3. Choose the name of the function that you just created in the Lambda console. The ARN of the Lambda function displays.
4. (Optional) For Tags, add key-value pairs to associate with this data source. For more information about tags, see Tagging Athena resources (p. 658).
5. Choose Next.
6. On the Review and create page, review the data source details, and then choose Create data source.
7. The Data source details section of the page for your data source shows information about your new connector. You can now use the connector in your Athena queries.

For information about writing queries with data connectors, see Writing federated queries (p. 167).

Using the AWS Serverless Application Repository to deploy a data source connector

To deploy a data source connector, you can use the AWS Serverless Application Repository instead of starting with the Athena console. Use the AWS Serverless Application Repository to find the connector that you want to use, provide the parameters that the connector requires, and then deploy the connector to your account. Then, after you deploy the connector, you use the Athena console to make the data source available to Athena.

Deploying the connector to Your Account

Note
To use the Amazon Athena Federated Query feature, set your workgroup to Athena engine version 2. For steps, see Changing Athena engine versions (p. 669).

To use the AWS Serverless Application Repository to deploy a data source connector to your account

1. Sign in to the AWS Management Console and open the Serverless App Repository.
2. In the navigation pane, choose Available applications.
3. Select the option **Show apps that create custom IAM roles or resource policies**.
4. In the search box, type the name of the connector. For a list of prebuilt Athena data connectors, see Using Athena data source connectors (p. 67).
5. Choose the name of the connector. Choosing a connector opens the Lambda function’s **Application details** page in the AWS Lambda console.
6. On the right side of the details page, for **Application settings**, fill in the required information. The minimum required settings include the following. For information about the remaining configurable options for data connectors built by Athena, see the corresponding **Available connectors** topic on GitHub.
   - **AthenaCatalogName** – A name for the Lambda function in lower case that indicates the data source that it targets, such as `cloudwatchlogs`.
   - **SpillBucket** – Specify an Amazon S3 bucket in your account to receive data from any large response payloads that exceed Lambda function response size limits.
7. Select **I acknowledge that this app creates custom IAM roles and resource policies**. For more information, choose the **Info** link.
8. At the bottom right of the **Application settings** section, choose **Deploy**. When the deployment is complete, the Lambda function appears in the **Resources** section in the Lambda console.

### Making the connector available in Athena

Now you are ready to use the Athena console to make the data source connector available to Athena.

**To make the data source connector available to Athena**

2. If the console navigation pane is not visible, choose the expansion menu on the left.
3. In the navigation pane, choose **Data sources**.
4. On the **Data sources** page, choose **Create data source**.
5. For **Choose a data source**, choose the data source for which you created a connector in the AWS Serverless Application Repository. This tutorial uses **Amazon CloudWatch Logs** as the federated data source.
6. Choose **Next**.
7. On the **Enter data source details** page, for **Data source name**, enter the name that you want to use in your SQL statements when you query the data source from Athena (for example, `CloudWatchLogs`). The name can be up to 127 characters and must be unique within your account. It cannot be changed after you create it. Valid characters are a-z, A-Z, 0-9, _ (underscore), @ (at sign) and - (hyphen). The names `awsdatacatalog`, `hive`, `jmx`, and `system` are reserved by Athena and cannot be used for data source names.
8. In the **Connection details** section, use the **Select or enter a Lambda function** box to choose the name of the function that you just created. The ARN of the Lambda function displays.
9. (Optional) For Tags, add key-value pairs to associate with this data source. For more information about tags, see Tagging Athena resources (p. 658).
10. Choose Next.
11. On the Review and create page, review the data source details, and then choose Create data source.
12. The Data source details section of the page for your data source shows information about your new connector. You can now use the connector in your Athena queries.

For information about writing queries with data connectors, see Writing federated queries (p. 167).

Creating a VPC for a data source connector

Some Athena data source connectors require a VPC and a security group. This topic shows you how to create a VPC with a subnet and a security group for the VPC. As part of this process, you retrieve the IDs for the VPC, subnet, and security group that you create. These IDs are required when you configure your connector for use with Athena.

To create a VPC for an Athena data source connector

1. Sign in to the AWS Management Console and open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. In the navigation pane, ensure that New VPC Experience is selected.
3. Choose Create VPC.
4. Under VPC Settings, for Resources to create, choose VPC and more.
5. For Auto-generate, enter a value that will be used to generate name tags for all resources in your VPC.
6. Choose Create VPC.
7. Choose View VPC.
8. In the Details section, for VPC ID, copy your VPC ID for later reference.

Now you are ready to retrieve the subnet ID for the VPC that you just created.

To retrieve your VPC subnet ID

1. In the VPC console navigation pane, choose Subnets.
2. Choose the name corresponding to the subnet that you created.
3. In the Details section, for Subnet ID, copy your subnet ID for later reference.

Next, you create a security group for your VPC.

To create a security group for your VPC

1. In the VPC console navigation pane, choose Security, Security Groups.
2. Choose Create security group.
3. On the Create security group page, enter the following information:
   - For Security group name, enter a name for your security group.
   - For Description, enter a description for the security group. This field is required.
   - For VPC, enter the VPC ID of the VPC that you created for your data source connector.
   - For Inbound rules and Outbound rules, add any inbound and outbound rules that you require.
4. Choose Create security group.
5. On the Details page for the security group, copy the Security group ID for later reference.

Using Athena data source connectors

This section lists prebuilt Athena data source connectors that you can use to query a variety of data sources external to Amazon S3. To use a connector in your Athena queries, configure it and deploy it to your account.

Considerations and limitations

- Some prebuilt connectors require that you create a VPC and a security group before you can use the connector. For information about creating VPCs, see Creating a VPC for a data source connector (p. 66).
- To use the Athena Federated Query feature with AWS Secrets Manager, you must configure an Amazon VPC private endpoint for Secrets Manager. For more information, see Create a Secrets Manager VPC private endpoint in the AWS Secrets Manager User Guide.
- Queries that include a predicate take significantly longer to execute. For small datasets, very little data is scanned, and queries take an average of about 2 minutes. However, for large datasets, many queries can time out.

Additional information

- For information about deploying an Athena data source connector, see Deploying a connector and connecting to a data source (p. 62).
- For information about writing queries that use Athena data source connectors, see Writing federated queries (p. 167).
- For in-depth information about the Athena data source connectors, see Available connectors on GitHub.

Athena data source connectors

- Amazon Athena Azure Data Lake Storage (ADLS) Gen2 connector (p. 68)
- Amazon Athena Azure Synapse connector (p. 72)
- Amazon Athena Google BigQuery connector (p. 77)
- Amazon Athena Cloudera Hive connector (p. 78)
- Amazon Athena Cloudera Impala connector (p. 83)
- Amazon Athena CloudWatch connector (p. 87)
- Amazon Athena CloudWatch Metrics connector (p. 89)
- Amazon Athena AWS CMDB connector (p. 91)
- Amazon Athena IBM Db2 connector (p. 93)
- Amazon Athena DocumentDB connector (p. 97)
- Amazon Athena DynamoDB connector (p. 101)
- Amazon Athena HBase connector (p. 104)
- Amazon Athena Hortonworks connector (p. 108)
- Amazon Athena MySQL connector (p. 113)
- Amazon Athena Neptune connector (p. 117)
- Amazon Athena OpenSearch connector (p. 118)
- Amazon Athena Oracle connector (p. 122)
- Amazon Athena PostgreSQL connector (p. 127)
• Amazon Athena Redis connector (p. 131)
• Amazon Athena Redshift connector (p. 133)
• Amazon Athena SAP HANA connector (p. 138)
• Amazon Athena Snowflake connector (p. 143)
• Amazon Athena Microsoft SQL Server connector (p. 148)
• Amazon Athena Teradata connector (p. 152)
• Amazon Athena Timescale connector (p. 157)
• Amazon Athena TPC benchmark DS (TPC-DS) connector (p. 160)
• Amazon Athena Vertica connector (p. 163)

Note
The AthenaJdbcConnector (latest version 2022.4.1) has been deprecated. Instead, use a database-specific connector like those for MySQL (p. 113), Redshift (p. 133), or PostgreSQL (p. 127).

Amazon Athena Azure Data Lake Storage (ADLS) Gen2 connector

The Amazon Athena connector for Azure Data Lake Storage (ADLS) Gen2 enables Amazon Athena to run SQL queries on data stored on ADLS.

Prerequisites

• Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).
• Set up a VPC and a security group before you use this connector. For more information, see Creating a VPC for a data source connector (p. 66).

Limitations

• Write DDL operations are not supported.
• In a multiplexer setup, the spill bucket and prefix are shared across all database instances.
• Any relevant Lambda limits. For more information, see Lambda quotas in the AWS Lambda Developer Guide.
• Date and timestamp data types in filter conditions must be cast to appropriate data types.

Terms

The following terms relate to the ADLS Gen2 connector.

• Database instance – Any instance of a database deployed on premises, on Amazon EC2, or on Amazon RDS.
• Handler – A Lambda handler that accesses your database instance. A handler can be for metadata or for data records.
• Metadata handler – A Lambda handler that retrieves metadata from your database instance.
• Record handler – A Lambda handler that retrieves data records from your database instance.
• Composite handler – A Lambda handler that retrieves both metadata and data records from your database instance.
- **Property or parameter** – A database property used by handlers to extract database information. You configure these properties as Lambda environment variables.

- **Connection String** – A string of text used to establish a connection to a database instance.

- **Catalog** – A non-AWS Glue catalog registered with Athena that is a required prefix for the connection_string property.

- **Multiplexing handler** – A Lambda handler that can accept and use multiple database connections.

## Parameters

Use the Lambda environment variables in this section to configure the ADLS Gen2 connector.

### Connection string

Use a JDBC connection string in the following format to connect to a database instance.

```
datalakegentwo://${jdbc_connection_string}
```

### Using a multiplexing handler

You can use a multiplexer to connect to multiple database instances with a single Lambda function. Requests are routed by catalog name. Use the following classes in Lambda.

<table>
<thead>
<tr>
<th>Handler</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>DataLakeGen2MuxCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>DataLakeGen2MuxMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>DataLakeGen2MuxRecordHandler</td>
</tr>
</tbody>
</table>

### Multiplexing handler parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$catalog_connection_string</td>
<td>Required. A database instance connection string. Prefix the string with the name of the catalog used in Athena. For example, if the catalog registered with Athena is mydatalakegentwocatalog, then the environment variable name is mydatalakegentwocatalog_connection_string.</td>
</tr>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when the catalog is lambda:${AWS_LAMBDA_FUNCTION_NAME}.</td>
</tr>
</tbody>
</table>

The following example properties are for a DataLakeGen2 MUX Lambda function that supports two database instances: datalakegentwo1 (the default), and datalakegentwo2.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>datalakegentwo://jdbc:sqlserver://adlsgentwo1.hostname:port;dbname=database_name;${secret1_name}</td>
</tr>
</tbody>
</table>
Providing credentials

To provide a user name and password for your database in your JDBC connection string, you can use connection string properties or AWS Secrets Manager.

- **Connection String** – A user name and password can be specified as properties in the JDBC connection string.
- **AWS Secrets Manager** – To use the Athena Federated Query feature with AWS Secrets Manager, the VPC connected to your Lambda function should have internet access or a VPC endpoint to connect to Secrets Manager.

You can put the name of a secret in AWS Secrets Manager in your JDBC connection string. The connector replaces the secret name with the username and password values from Secrets Manager.

For Amazon RDS database instances, this support is tightly integrated. If you use Amazon RDS, we highly recommend using AWS Secrets Manager and credential rotation. If your database does not use Amazon RDS, store the credentials as JSON in the following format:

```
{"username": "${username}", "password": "${password}"
```

**Example connection string with secret name**

The following string has the secret name `secret1_name`.

```
datalakegentwo://jdbc:sqlserver://hostname:port;databaseName=database_name;${secret1_name}
```

The connector uses the secret name to retrieve secrets and provide the user name and password, as in the following example.

```
datalakegentwo://jdbc:sqlserver://hostname:port;databaseName=database_name;user=user_name;password=password
```

**Using a single connection handler**

You can use the following single connection metadata and record handlers to connect to a single ADLS Gen2 instance.

<table>
<thead>
<tr>
<th>Handler type</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>DataLakeGen2CompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>DataLakeGen2MetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>DataLakeGen2RecordHandler</td>
</tr>
</tbody>
</table>
Single connection handler parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when a catalog is not recognized.</td>
</tr>
</tbody>
</table>

The single connection handlers support one database instance and must provide a default connection string parameter. All other connection strings are ignored.

The following example property is for a single ADLS Gen2 instance supported by a Lambda function.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>datalakegentwo://jdbc:sqlserver://hostname:port;databaseName=;${secret_name}</td>
</tr>
</tbody>
</table>

Spill parameters

The Lambda SDK can spill data to Amazon S3. All database instances accessed by the same Lambda function spill to the same location.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spill_bucket</td>
<td>Required. Spill bucket name.</td>
</tr>
<tr>
<td>spill_prefix</td>
<td>Required. Spill bucket key prefix.</td>
</tr>
<tr>
<td>spill_put_request_headers</td>
<td>(Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, &quot;x-amz-server-side-encryption&quot; : &quot;AES256&quot;). For other possible headers, see PutObject in the Amazon Simple Storage Service API Reference.</td>
</tr>
</tbody>
</table>

Data type support

The following table shows the corresponding data types for ADLS Gen2 and Arrow.

<table>
<thead>
<tr>
<th>ADLS Gen2</th>
<th>Arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>TINYINT</td>
</tr>
<tr>
<td>tinyint</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>smallint</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>int</td>
<td>INT</td>
</tr>
<tr>
<td>bigint</td>
<td>BIGINT</td>
</tr>
<tr>
<td>decimal</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>numeric</td>
<td>FLOAT8</td>
</tr>
<tr>
<td>ADLS Gen2</td>
<td>Arrow</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------</td>
</tr>
<tr>
<td>smallmoney</td>
<td>FLOAT8</td>
</tr>
<tr>
<td>money</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>float[24]</td>
<td>FLOAT4</td>
</tr>
<tr>
<td>float[53]</td>
<td>FLOAT8</td>
</tr>
<tr>
<td>real</td>
<td>FLOAT4</td>
</tr>
<tr>
<td>datetime</td>
<td>Date(MILLISECOND)</td>
</tr>
<tr>
<td>datetime2</td>
<td>Date(MILLISECOND)</td>
</tr>
<tr>
<td>smalldatetime</td>
<td>Date(MILLISECOND)</td>
</tr>
<tr>
<td>date</td>
<td>Date(DAY)</td>
</tr>
<tr>
<td>time</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>datetimeoffset</td>
<td>Date(MILLISECOND)</td>
</tr>
<tr>
<td>char[n]</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>varchar[n/max]</td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>

**Partitions and splits**

Azure Data Lake Storage Gen2 uses Hadoop compatible Gen2 blob storage for storing data files. The data from these files is queried from the Azure Synapse engine. The Azure Synapse engine treats Gen2 data stored in file systems as external tables. The partitions are implemented based on the type of data. If the data has already been partitioned and distributed within the Gen 2 storage system, the connector retrieves the data as single split.

**License information**

By using this connector, you acknowledge the inclusion of third party components, a list of which can be found in the pom.xml file for this connector, and agree to the terms in the respective third party licenses provided in the LICENSE.txt file on GitHub.com.

**See also**

For the latest JDBC driver version information, see the pom.xml file for the ADLS Gen2 connector on GitHub.com.

For additional information about this connector, visit the corresponding site on GitHub.com.

**Amazon Athena Azure Synapse connector**

The Amazon Athena connector for Azure Synapse analytics enables Amazon Athena to run SQL queries on your Azure Synapse databases using JDBC.

**Prerequisites**

- Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a
Limitations

- Write DDL operations are not supported.
- In a multiplexer setup, the spill bucket and prefix are shared across all database instances.
- Any relevant Lambda limits. For more information, see Lambda quotas in the AWS Lambda Developer Guide.
- In filter conditions, you must cast the Date and Timestamp data types to the appropriate data type.
- To search for negative values of type Real and Float, use the <= or >= operator.
- The binary, varbinary, image, and rowversion data types are not supported.

Terms

The following terms relate to the Synapse connector.

- **Database instance** – Any instance of a database deployed on premises, on Amazon EC2, or on Amazon RDS.
- **Handler** – A Lambda handler that accesses your database instance. A handler can be for metadata or for data records.
- **Metadata handler** – A Lambda handler that retrieves metadata from your database instance.
- **Record handler** – A Lambda handler that retrieves data records from your database instance.
- **Composite handler** – A Lambda handler that retrieves both metadata and data records from your database instance.
- **Property or parameter** – A database property used by handlers to extract database information. You configure these properties as Lambda environment variables.
- **Connection String** – A string of text used to establish a connection to a database instance.
- **Catalog** – A non-AWS Glue catalog registered with Athena that is a required prefix for the connection_string property.
- **Multiplexing handler** – A Lambda handler that can accept and use multiple database connections.

Parameters

Use the Lambda environment variables in this section to configure the Synapse connector.

**Connection string**

Use a JDBC connection string in the following format to connect to a database instance.

```
synapse://${jdbc_connection_string}
```

**Using a multiplexing handler**

You can use a multiplexer to connect to multiple database instances with a single Lambda function. Requests are routed by catalog name. Use the following classes in Lambda.
### Athena data source connectors

<table>
<thead>
<tr>
<th>Handler</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>SynapseMuxCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>SynapseMuxMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>SynapseMuxRecordHandler</td>
</tr>
</tbody>
</table>

#### Multiplexing handler parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$catalog_connection_string</td>
<td>Required. A database instance connection string. Prefix the string with the name of the catalog used in Athena. For example, if the catalog registered with Athena is <code>mysynapsecatalog</code>, then the environment variable name is <code>mysynapsecatalog_connection_string</code>.</td>
</tr>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when the catalog is lambda:<code>${AWS_LAMBDA_FUNCTION_NAME}</code>.</td>
</tr>
</tbody>
</table>

The following example properties are for a Synapse MUX Lambda function that supports two database instances: `synapse1` (the default), and `synapse2`.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>synapse://jdbc:synapse://synapse1.hostname:port;databaseName=(&lt;database_name&gt;);=${secret1_name}</td>
</tr>
<tr>
<td>synapse_catalog1_connection_string</td>
<td>synapse://jdbc:synapse://synapse1.hostname:port;databaseName=(&lt;database_name&gt;);=${secret1_name}</td>
</tr>
<tr>
<td>synapse_catalog2_connection_string</td>
<td>synapse://jdbc:synapse://synapse2.hostname:port;databaseName=(&lt;database_name&gt;);=${secret2_name}</td>
</tr>
</tbody>
</table>

#### Providing credentials

To provide a user name and password for your database in your JDBC connection string, you can use connection string properties or AWS Secrets Manager.

- **Connection String** – A user name and password can be specified as properties in the JDBC connection string.
- **AWS Secrets Manager** – To use the Athena Federated Query feature with AWS Secrets Manager, the VPC connected to your Lambda function should have internet access or a VPC endpoint to connect to Secrets Manager.

You can put the name of a secret in AWS Secrets Manager in your JDBC connection string. The connector replaces the secret name with the username and password values from Secrets Manager.

For Amazon RDS database instances, this support is tightly integrated. If you use Amazon RDS, we highly recommend using AWS Secrets Manager and credential rotation. If your database does not use Amazon RDS, store the credentials as JSON in the following format:
Example connection string with secret name

The following string has the secret name ${secret_name}.

```
synapse://jdbc:synapse://hostname:port;databaseName=<database_name>${secret_name}
```

The connector uses the secret name to retrieve secrets and provide the user name and password, as in the following example.

```
synapse://jdbc:synapse://hostname:port;databaseName=<database_name>;user=<user>;password=<password>
```

Using a single connection handler

You can use the following single connection metadata and record handlers to connect to a single Synapse instance.

<table>
<thead>
<tr>
<th>Handler type</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>SynapseCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>SynapseMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>SynapseRecordHandler</td>
</tr>
</tbody>
</table>

Single connection handler parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when a catalog is not recognized.</td>
</tr>
</tbody>
</table>

The single connection handlers support one database instance and must provide a default connection string parameter. All other connection strings are ignored.

The following example property is for a single Synapse instance supported by a Lambda function.

```
default synapse://jdbc:sqlserver://hostname:port;databaseName=<database_name>;${secret_name}
```

Spill parameters

The Lambda SDK can spill data to Amazon S3. All database instances accessed by the same Lambda function spill to the same location.
### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spill_bucket</td>
<td>Required. Spill bucket name.</td>
</tr>
<tr>
<td>spill_prefix</td>
<td>Required. Spill bucket key prefix.</td>
</tr>
<tr>
<td>spill_put_request_headers</td>
<td>(Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, <code>{&quot;x-amz-server-side-encryption&quot;: &quot;AES256&quot;}</code>). For other possible headers, see <a href="https://docs.aws.amazon.com/AmazonS3/latest/API/RESTObjectPut.html">PutObject</a> in the Amazon Simple Storage Service API Reference.</td>
</tr>
</tbody>
</table>

### Data type support

The following table shows the corresponding data types for Synapse and Apache Arrow.

<table>
<thead>
<tr>
<th>Synapse</th>
<th>Arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>TINYINT</td>
</tr>
<tr>
<td>tinyint</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>smallint</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>int</td>
<td>INT</td>
</tr>
<tr>
<td>bigint</td>
<td>BIGINT</td>
</tr>
<tr>
<td>decimal</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>numeric</td>
<td>FLOAT8</td>
</tr>
<tr>
<td>smallmoney</td>
<td>FLOAT8</td>
</tr>
<tr>
<td>money</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>float[24]</td>
<td>FLOAT4</td>
</tr>
<tr>
<td>float[53]</td>
<td>FLOAT8</td>
</tr>
<tr>
<td>real</td>
<td>FLOAT4</td>
</tr>
<tr>
<td>datetime</td>
<td>Date(MILLISECOND)</td>
</tr>
<tr>
<td>datetime2</td>
<td>Date(MILLISECOND)</td>
</tr>
<tr>
<td>smalldatetime</td>
<td>Date(MILLISECOND)</td>
</tr>
<tr>
<td>date</td>
<td>Date(DAY)</td>
</tr>
<tr>
<td>time</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>datetimeoffset</td>
<td>Date(MILLISECOND)</td>
</tr>
<tr>
<td>char[n]</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>varchar[n/max]</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>nchar[n]</td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>
Synapse | Arrow
---|---
nvarchar[n/max] | VARCHAR

**Partitions and splits**

A partition is represented by a single partition column of type `varchar`. Synapse supports range partitioning, so partitioning is implemented by extracting the partition column and partition range from Synapse metadata tables. These range values are used to create the splits.

**License information**

By using this connector, you acknowledge the inclusion of third party components, a list of which can be found in the `pom.xml` file for this connector, and agree to the terms in the respective third party licenses provided in the `LICENSE.txt` file on GitHub.com.

**See also**

For the latest JDBC driver version information, see the `pom.xml` file for the Synapse connector on GitHub.com.

For additional information about this connector, visit the corresponding site on GitHub.com.

**Amazon Athena Google BigQuery connector**

The Amazon Athena connector for Google BigQuery enables Amazon Athena to run SQL queries on your Google BigQuery data.

**Prerequisites**

- Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).
- Set up a VPC and a security group before you use this connector. For more information, see Creating a VPC for a data source connector (p. 66).

**Limitations**

- Lambda functions have a maximum timeout value of 15 minutes. Each split executes a query on BigQuery and must finish with enough time to store the results for Athena to read. If the Lambda function times out, the query fails.
- Google BigQuery is case sensitive. The connector attempts to correct the case of dataset names and table names but does not do any case correction for project IDs. This is necessary because Athena lower cases all metadata. These corrections make many extra calls to Google BigQuery.
- Binary data types are not supported.
- Complex data types such as `map`, `list`, and `struct` are not supported.
- Because of Google BigQuery concurrency and quota limits, the connector may encounter Google quota limit issues. To avoid these issues, push as many constraints to Google BigQuery as feasible. For information about BigQuery quotas, see Quotas and limits in the Google BigQuery documentation.

**Parameters**

Use the Lambda environment variables in this section to configure the Google BigQuery connector.
• **spill_bucket** – Specifies the Amazon S3 bucket for data that exceeds Lambda function limits.

• **spill_prefix** – (Optional) Defaults to a subfolder in the specified spill_bucket called athena-federation-spill. We recommend that you configure an Amazon S3 storage lifecycle on this location to delete spills older than a predetermined number of days or hours.

• **spill_put_request_headers** – (Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, {"x-amz-server-side-encryption" : "AES256"}). For other possible headers, see PutObject in the Amazon Simple Storage Service API Reference.

• **kms_key_id** – (Optional) By default, any data that is spilled to Amazon S3 is encrypted using the AES-GCM authenticated encryption mode and a randomly generated key. To have your Lambda function use stronger encryption keys generated by KMS like a7e63k4b-8loc-40db-a2a1-4d0en2cd8331, you can specify a KMS key ID.

• **disable_spill_encryption** – (Optional) When set to True, disables spill encryption. Defaults to False so that data that is spilled to S3 is encrypted using AES-GCM – either using a randomly generated key or KMS to generate keys. Disabling spill encryption can improve performance, especially if your spill location uses server-side encryption.

• **gcp_project_id** – The project ID (not project name) that contains the datasets that the connector should read from (for example, semiotic-primer-1234567).

• **secret_manager_gcp_creds_name** – The name of the secret within AWS Secrets Manager that contains your BigQuery credentials in JSON format (for example, GoogleCloudPlatformCredentials).

### Partitions and splits

The BigQuery connector uses the concurrencyLimit environment variable that is defined in your Google project to determine the page count for the splits. The connector does not generate splits as a function of the partitions over the table.

### Performance

In order to decrease the number of results that must be returned, the connector attempts to push as many constraints to Google BigQuery as possible.

### License information

The Amazon Athena Google BigQuery connector project is licensed under the Apache-2.0 License.

By using this connector, you acknowledge the inclusion of third party components, a list of which can be found in the pom.xml file for this connector, and agree to the terms in the respective third party licenses provided in the LICENSE.txt file on GitHub.com.

### See also

For additional information about this connector, visit the corresponding site on GitHub.com.

## Amazon Athena Cloudera Hive connector

The Amazon Athena connector for Cloudera Hive enables Athena to run SQL queries on the Cloudera Hive Hadoop distribution. The connector transforms your Athena SQL queries to their equivalent HiveQL syntax.

### Prerequisites

- Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a
data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).

- Set up a VPC and a security group before you use this connector. For more information, see Creating a VPC for a data source connector (p. 66).

**Limitations**

- Write DDL operations are not supported.
- In a multiplexer setup, the spill bucket and prefix are shared across all database instances.
- Any relevant Lambda limits. For more information, see Lambda quotas in the AWS Lambda Developer Guide.

**Terms**

The following terms relate to the Cloudera Hive connector.

- **Database instance** – Any instance of a database deployed on premises, on Amazon EC2, or on Amazon RDS.
- **Handler** – A Lambda handler that accesses your database instance. A handler can be for metadata or for data records.
- **Metadata handler** – A Lambda handler that retrieves metadata from your database instance.
- **Record handler** – A Lambda handler that retrieves data records from your database instance.
- **Composite handler** – A Lambda handler that retrieves both metadata and data records from your database instance.
- **Property or parameter** – A database property used by handlers to extract database information. You configure these properties as Lambda environment variables.
- **Connection String** – A string of text used to establish a connection to a database instance.
- **Catalog** – A non-AWS Glue catalog registered with Athena that is a required prefix for the connection_string property.
- **Multiplexing handler** – A Lambda handler that can accept and use multiple database connections.

**Parameters**

Use the Lambda environment variables in this section to configure the Cloudera Hive connector.

**Connection string**

Use a JDBC connection string in the following format to connect to a database instance.

```sql
hive://${jdbc_connection_string}
```

**Using a multiplexing handler**

You can use a multiplexer to connect to multiple database instances with a single Lambda function. Requests are routed by catalog name. Use the following classes in Lambda.

<table>
<thead>
<tr>
<th>Handler</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>HiveMuxCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>HiveMuxMetadataHandler</td>
</tr>
</tbody>
</table>
### Handler

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record handler</td>
<td>HiveMuxRecordHandler</td>
</tr>
</tbody>
</table>

### Multiplexing handler parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$catalog_connection_string</td>
<td>Required. A database instance connection string. Prefix the string with the name of the catalog used in Athena. For example, if the catalog registered with Athena is myhivecatalog, then the environment variable name is myhivecatalog_connection_string.</td>
</tr>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when the catalog is lambda:${AWS_LAMBDA_FUNCTION_NAME}.</td>
</tr>
</tbody>
</table>

The following example properties are for a Hive MUX Lambda function that supports two database instances: hive1 (the default), and hive2.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>hive://jdbc:hive2://hive1:10000/default?${Test/RDS/hive1}</td>
</tr>
<tr>
<td>hive2_catalog1_connection_string</td>
<td>hive://jdbc:hive2://hive1:10000/default?${Test/RDS/hive1}</td>
</tr>
</tbody>
</table>

### Providing credentials

To provide a user name and password for your database in your JDBC connection string, you can use connection string properties or AWS Secrets Manager.

- **Connection String** – A user name and password can be specified as properties in the JDBC connection string.
- **AWS Secrets Manager** – To use the Athena Federated Query feature with AWS Secrets Manager, the VPC connected to your Lambda function should have internet access or a VPC endpoint to connect to Secrets Manager.

You can put the name of a secret in AWS Secrets Manager in your JDBC connection string. The connector replaces the secret name with the username and password values from Secrets Manager.

For Amazon RDS database instances, this support is tightly integrated. If you use Amazon RDS, we highly recommend using AWS Secrets Manager and credential rotation. If your database does not use Amazon RDS, store the credentials as JSON in the following format:

```json
{"username": "$(username)", "password": "$(password)"}
```

### Example connection string with secret name

The following string has the secret name ${Test/RDS/hive1}.
The connector uses the secret name to retrieve secrets and provide the user name and password, as in the following example.

```
hive://jdbc:hive2://hive1:10000/default?...&${Test/RDS/hive1}&...
```

Currently, the Cloudera Hive connector recognizes the $UID$ and $PWD$ JDBC properties.

**Using a single connection handler**

You can use the following single connection metadata and record handlers to connect to a single Cloudera Hive instance.

<table>
<thead>
<tr>
<th>Handler type</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>HiveCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>HiveMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>HiveRecordHandler</td>
</tr>
</tbody>
</table>

**Single connection handler parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when a catalog is not recognized.</td>
</tr>
</tbody>
</table>

The single connection handlers support one database instance and must provide a `default` connection string parameter. All other connection strings are ignored.

The following example property is for a single Cloudera Hive instance supported by a Lambda function.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>hive://jdbc:hive2://hive1:10000/default?secret=${Test/RDS/hive1}</td>
</tr>
</tbody>
</table>

**Spill parameters**

The Lambda SDK can spill data to Amazon S3. All database instances accessed by the same Lambda function spill to the same location.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spill_bucket</td>
<td>Required. Spill bucket name.</td>
</tr>
<tr>
<td>spill_prefix</td>
<td>Required. Spill bucket key prefix.</td>
</tr>
<tr>
<td>spill_put_request_headers</td>
<td>(Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, {&quot;x-amz-server-side-encryption&quot;:}</td>
</tr>
</tbody>
</table>
Parameter | Description
---|---
"AES256" | For other possible headers, see `PutObject` in the Amazon Simple Storage Service API Reference.

Data type support

The following table shows the corresponding data types for JDBC, Cloudera Hive, and Arrow.

<table>
<thead>
<tr>
<th>JDBC</th>
<th>Cloudera Hive</th>
<th>Arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Boolean</td>
<td>Bit</td>
</tr>
<tr>
<td>Integer</td>
<td>TINYINT</td>
<td>Tiny</td>
</tr>
<tr>
<td>Short</td>
<td>SMALLINT</td>
<td>Smallint</td>
</tr>
<tr>
<td>Integer</td>
<td>INT</td>
<td>Int</td>
</tr>
<tr>
<td>Long</td>
<td>BIGINT</td>
<td>Bigint</td>
</tr>
<tr>
<td>float</td>
<td>float4</td>
<td>Float4</td>
</tr>
<tr>
<td>Double</td>
<td>float8</td>
<td>Float8</td>
</tr>
<tr>
<td>Date</td>
<td>date</td>
<td>DateDay</td>
</tr>
<tr>
<td>Timestamp</td>
<td>timestamp</td>
<td>DateMilli</td>
</tr>
<tr>
<td>String</td>
<td>VARCHAR</td>
<td>Varchar</td>
</tr>
<tr>
<td>Bytes</td>
<td>bytes</td>
<td>Varbinary</td>
</tr>
<tr>
<td>BigDecimal</td>
<td>Decimal</td>
<td>Decimal</td>
</tr>
<tr>
<td>ARRAY</td>
<td>N/A (see note)</td>
<td>List</td>
</tr>
</tbody>
</table>

**Note**
Currently, Cloudera Hive does not support the aggregate types ARRAY, MAP, STRUCT, or UNIONTYPE. Columns of aggregate types are treated as VARCHAR columns in SQL.

Partitions and splits

Partitions are used to determine how to generate splits for the connector. Athena constructs a synthetic column of type `varchar` that represents the partitioning scheme for the table to help the connector generate splits. The connector does not modify the actual table definition.

Performance tuning

Cloudera Hive supports static partitions. The Athena Lambda connector can retrieve data from these partitions in parallel. If you want to query very large datasets with uniform partition distribution, static partitioning is highly recommended.

License information

By using this connector, you acknowledge the inclusion of third party components, a list of which can be found in the `pom.xml` file for this connector, and agree to the terms in the respective third party licenses provided in the `LICENSE.txt` file on GitHub.com.
See also

For the latest JDBC driver version information, see the pom.xml file for the Cloudera Hive connector on GitHub.com.

For additional information about this connector, visit the corresponding site on GitHub.com.

Amazon Athena Cloudera Impala connector

The Amazon Athena Cloudera Impala connector enables Athena to run SQL queries on the Cloudera Impala Hadoop distribution. The connector transforms your Athena SQL queries to their equivalent Impala syntax.

Prerequisites

- Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).
- Set up a VPC and a security group before you use this connector. For more information, see Creating a VPC for a data source connector (p. 66).

Limitations

- Write DDL operations are not supported.
- In a multiplexer setup, the spill bucket and prefix are shared across all database instances.
- Any relevant Lambda limits. For more information, see Lambda quotas in the AWS Lambda Developer Guide.

Terms

The following terms relate to the Cloudera Impala connector.

- **Database instance** – Any instance of a database deployed on premises, on Amazon EC2, or on Amazon RDS.
- **Handler** – A Lambda handler that accesses your database instance. A handler can be for metadata or data records.
- **Metadata handler** – A Lambda handler that retrieves metadata from your database instance.
- **Record handler** – A Lambda handler that retrieves data records from your database instance.
- **Composite handler** – A Lambda handler that retrieves both metadata and data records from your database instance.
- **Property or parameter** – A database property used by handlers to extract database information. You configure these properties as Lambda environment variables.
- **Connection String** – A string of text used to establish a connection to a database instance.
- **Catalog** – A non-AWS Glue catalog registered with Athena that is a required prefix for the connection_string property.
- **Multiplexing handler** – A Lambda handler that can accept and use multiple database connections.

Parameters

Use the Lambda environment variables in this section to configure the Cloudera Impala connector.
Connection string

Use a JDBC connection string in the following format to connect to a database instance.

```
impala://${jdbc_connection_string}
```

Using a multiplexing handler

You can use a multiplexer to connect to multiple database instances with a single Lambda function. Requests are routed by catalog name. Use the following classes in Lambda.

<table>
<thead>
<tr>
<th>Handler</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>ImpalaMuxCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>ImpalaMuxMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>ImpalaMuxRecordHandler</td>
</tr>
</tbody>
</table>

Multiplexing handler parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$catalog_connection_string</code></td>
<td>Required. A database instance connection string. Prefix the string with the name of the catalog used in Athena. For example, if the catalog registered with Athena is myimpalacatalog, then the environment variable name is myimpalacatalog_connection_string.</td>
</tr>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when the catalog is lambda:[$AWS_LAMBDA_FUNCTION_NAME].</td>
</tr>
</tbody>
</table>

The following example properties are for a Impala MUX Lambda function that supports two database instances: impala1 (the default), and impala2.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>impala://jdbc:impala2://impala1:10000/default?${Test/RDS/impala1}</td>
</tr>
<tr>
<td>impala_catalog1_connection_string</td>
<td>impala://jdbc:impala2://impala1:10000/default?${Test/RDS/impala1}</td>
</tr>
</tbody>
</table>

Providing credentials

To provide a user name and password for your database in your JDBC connection string, you can use connection string properties or AWS Secrets Manager.

- **Connection String** – A user name and password can be specified as properties in the JDBC connection string.
Athena data source connectors

- **AWS Secrets Manager** – To use the Athena Federated Query feature with AWS Secrets Manager, the VPC connected to your Lambda function should have internet access or a VPC endpoint to connect to Secrets Manager.

  You can put the name of a secret in AWS Secrets Manager in your JDBC connection string. The connector replaces the secret name with the username and password values from Secrets Manager.

  For Amazon RDS database instances, this support is tightly integrated. If you use Amazon RDS, we highly recommend using AWS Secrets Manager and credential rotation. If your database does not use Amazon RDS, store the credentials as JSON in the following format:

  ```json
  {"username": "${username}"}, "password": "${password}"
  ```

  **Example connection string with secret name**

  The following string has the secret name `${Test/RDS/Impala1host}`.

  ```sql
  impala://jdbc:impala2://Impala1host:10000/default?...&${Test/RDS/Impala1host}&...
  ```

  The connector uses the secret name to retrieve secrets and provide the user name and password, as in the following example.

  ```sql
  impala://jdbc:impala2://Impala1host:10000/default?...&UID=sample2&PWD=sample2&...
  ```

  Currently, Cloudera Impala recognizes the UID and PWD JDBC properties.

  **Using a single connection handler**

  You can use the following single connection metadata and record handlers to connect to a single Cloudera Impala instance.

<table>
<thead>
<tr>
<th>Handler type</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>ImpalaCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>ImpalaMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>ImpalaRecordHandler</td>
</tr>
</tbody>
</table>

  **Single connection handler parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when a catalog is not recognized.</td>
</tr>
</tbody>
</table>

  The single connection handlers support one database instance and must provide a default connection string parameter. All other connection strings are ignored.

  The following example property is for a single Cloudera Impala instance supported by a Lambda function.
### Property | Value
---|---
default | `impala://jdbc:impala://Impalalhost:10000/default?secret=${Test/RDS/Impalalhost}`

**Spill parameters**

The Lambda SDK can spill data to Amazon S3. All database instances accessed by the same Lambda function spill to the same location.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spill_bucket</td>
<td>Required. Spill bucket name.</td>
</tr>
<tr>
<td>spill_prefix</td>
<td>Required. Spill bucket key prefix.</td>
</tr>
<tr>
<td>spill_put_request_headers</td>
<td>(Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, <code>{&quot;x-amz-server-side-encryption&quot; : &quot;AES256&quot;}</code>). For other possible headers, see <a href="https://docs.aws.amazon.com/AmazonS3/latest/API/api_get_object.html">PutObject</a> in the <a href="https">Amazon Simple Storage Service API Reference</a>.</td>
</tr>
</tbody>
</table>

**Data type support**

The following table shows the corresponding data types for JDBC, Cloudera Impala, and Arrow.

<table>
<thead>
<tr>
<th>JDBC</th>
<th>Cloudera Impala</th>
<th>Arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Boolean</td>
<td>Bit</td>
</tr>
<tr>
<td>Integer</td>
<td>TINYINT</td>
<td>Tiny</td>
</tr>
<tr>
<td>Short</td>
<td>SMALLINT</td>
<td>Smallint</td>
</tr>
<tr>
<td>Integer</td>
<td>INT</td>
<td>Int</td>
</tr>
<tr>
<td>Long</td>
<td>BIGINT</td>
<td>Bigint</td>
</tr>
<tr>
<td>float</td>
<td>float4</td>
<td>Float4</td>
</tr>
<tr>
<td>Double</td>
<td>float8</td>
<td>Float8</td>
</tr>
<tr>
<td>Date</td>
<td>date</td>
<td>DateMilli</td>
</tr>
<tr>
<td>Timestamp</td>
<td>timestamp</td>
<td>DateMilli</td>
</tr>
<tr>
<td>String</td>
<td>VARCHAR</td>
<td>Varchar</td>
</tr>
<tr>
<td>Bytes</td>
<td>bytes</td>
<td>Varbinary</td>
</tr>
<tr>
<td>BigDecimal</td>
<td>Decimal</td>
<td>Decimal</td>
</tr>
<tr>
<td>ARRAY</td>
<td>N/A (see note)</td>
<td>List</td>
</tr>
</tbody>
</table>
Note
Currently, Cloudera Impala does not support the aggregate types ARRAY, MAP, STRUCT, or UNIONTYPE. Columns of aggregate types are treated as VARCHAR columns in SQL.

Partitions and splits
Partitions are used to determine how to generate splits for the connector. Athena constructs a synthetic column of type VARCHAR that represents the partitioning scheme for the table to help the connector generate splits. The connector does not modify the actual table definition.

Performance tuning
Cloudera Impala supports static partitions. The Athena Lambda connector can retrieve data from these partitions in parallel. If you want to query very large datasets with uniform partition distribution, static partitioning is highly recommended.

License information
By using this connector, you acknowledge the inclusion of third party components, a list of which can be found in the pom.xml file for this connector, and agree to the terms in the respective third party licenses provided in the LICENSE.txt file on GitHub.com.

See also
For the latest JDBC driver version information, see the pom.xml file for the Cloudera Impala connector on GitHub.com.
For additional information about this connector, visit the corresponding site on GitHub.com.

Amazon Athena CloudWatch connector
The Amazon Athena CloudWatch connector enables Amazon Athena to communicate with CloudWatch so that you can query your log data with SQL.

The connector maps your LogGroups as schemas and each LogStream as a table. The connector also maps a special all_log_streams view that contains all LogStreams in the LogGroup. This view enables you to query all the logs in a LogGroup at once instead of searching through each LogStream individually.

Prerequisites

- Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).

Parameters
Use the Lambda environment variables in this section to configure the CloudWatch connector.

- **spill_bucket** – Specifies the Amazon S3 bucket for data that exceeds Lambda function limits.
- **spill_prefix** – (Optional) Defaults to a subfolder in the specified spill_bucket called athena-federation-spill. We recommend that you configure an Amazon S3 storage lifecycle on this location to delete spills older than a predetermined number of days or hours.
- **spill_put_request_headers** – (Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, {"x-amz-server-side-encryption" : "AES256"}). For other possible headers, see PutObject in the Amazon Simple Storage Service API Reference.
• **kms_key_id** – (Optional) By default, any data that is spilled to Amazon S3 is encrypted using the AES-GCM authenticated encryption mode and a randomly generated key. To have your Lambda function use stronger encryption keys generated by KMS like `a7e63k4b-8loc-40db-a2a1-4d0en2cd8331`, you can specify a KMS key ID.

• **disable_spill_encryption** – (Optional) When set to `True`, disables spill encryption. Defaults to `False` so that data that is spilled to S3 is encrypted using AES-GCM – either using a randomly generated key or KMS to generate keys. Disabling spill encryption can improve performance, especially if your spill location uses **server-side encryption**.

The connector also supports **AIMD congestion control** for handling throttling events from CloudWatch through the **Amazon Athena Query Federation SDK** ThrottlingInvoker construct. You can tweak the default throttling behavior by setting any of the following optional environment variables:

• **throttle_initial_delay_ms** – The initial call delay applied after the first congestion event. The default is 10 milliseconds.

• **throttle_max_delay_ms** – The maximum delay between calls. You can derive TPS by dividing it into 1000ms. The default is 1000 milliseconds.

• **throttle_decrease_factor** – The factor by which Athena reduces the call rate. The default is 0.5

• **throttle_increase_ms** – The rate at which Athena decreases the call delay. The default is 10 milliseconds.

**Databases and tables**

The Athena CloudWatch connector maps your LogGroups as schemas (that is, databases) and each LogStream as a table. The connector also maps a special `all_log_streams` view that contains all LogStreams in the LogGroup. This view enables you to query all the logs in a LogGroup at once instead of searching through each LogStream individually.

Every table mapped by the Athena CloudWatch connector has the following schema. This schema matches the fields provided by CloudWatch Logs.

• **log_stream** – A VARCHAR that contains the name of the LogStream that the row is from.

• **time** – An INT64 that contains the epoch time of when the log line was generated.

• **message** – A VARCHAR that contains the log message.

**Examples**

The following example shows how to perform a SELECT query on a specified LogStream.

```sql
SELECT *
FROM "lambda:cloudwatch_connector_lambda_name"."log_group_path"."log_stream_name"
LIMIT 100
```

The following example shows how to use the `all_log_streams` view to perform a query on all LogStreams in a specified LogGroup.

```sql
SELECT *
FROM "lambda:cloudwatch_connector_lambda_name"."log_group_path"."all_log_streams"
LIMIT 100
```

**Required Permissions**

Review the Policies section of the `athena-cloudwatch.yaml` file for full details on the IAM policies that this connector requires. The following is a brief summary.
• **Amazon S3 write access** – To successfully handle large queries, the connector requires write access to a location in Amazon S3.

• **Athena GetQueryExecution** – The connector uses this permission to fast-fail when the upstream Athena query has terminated.

• **CloudWatch Logs Read/Write** – The connector uses this permission to read your log data and to write its diagnostic logs.

**Performance tuning**

The Athena CloudWatch connector attempts to optimize queries against CloudWatch by parallelizing scans of the log streams required for your query. For certain time period filters, predicate pushdown is performed both within the Lambda function and within CloudWatch Logs.

**License information**

The Amazon Athena CloudWatch connector project is licensed under the [Apache-2.0 License](https://www.apache.org/licenses/LICENSE-2.0).

**See also**

For additional information about this connector, visit the corresponding site on GitHub.com.

**Amazon Athena CloudWatch Metrics connector**

The Amazon Athena CloudWatch Metrics connector enables Amazon Athena to communicate with CloudWatch Metrics so that you can query your metrics data with SQL.

**Prerequisites**

• Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).

**Parameters**

Use the Lambda environment variables in this section to configure the CloudWatch Metrics connector.

• **spill_bucket** – Specifies the Amazon S3 bucket for data that exceeds Lambda function limits.

• **spill_prefix** – (Optional) Defaults to a subfolder in the specified spill_bucket called athena-federation-spill. We recommend that you configure an Amazon S3 storage lifecycle on this location to delete spills older than a predetermined number of days or hours.

• **spill_put_request_headers** – (Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, `{"x-amz-server-side-encryption" : "AES256"}`). For other possible headers, see PutObject in the Amazon Simple Storage Service API Reference.

• **kms_key_id** – (Optional) By default, any data that is spilled to Amazon S3 is encrypted using the AES-GCM authenticated encryption mode and a randomly generated key. To have your Lambda function use stronger encryption keys generated by KMS like a7e63k4b-8loc-40db-a2a1-4d0en2cd8331, you can specify a KMS key ID.

• **disable_spill_encryption** – (Optional) When set to True, disables spill encryption. Defaults to False so that data that is spilled to S3 is encrypted using AES-GCM – either using a randomly generated key or KMS to generate keys. Disabling spill encryption can improve performance, especially if your spill location uses server-side encryption.
The connector also supports **AIMD congestion control** for handling throttling events from CloudWatch through the **Amazon Athena Query Federation SDK ThrottlingInvoker** construct. You can tweak the default throttling behavior by setting any of the following optional environment variables:

- **throttle_initial_delay_ms** – The initial call delay applied after the first congestion event. The default is 10 milliseconds.
- **throttle_max_delay_ms** – The maximum delay between calls. You can derive TPS by dividing it into 1000ms. The default is 1000 milliseconds.
- **throttle_decrease_factor** – The factor by which Athena reduces the call rate. The default is 0.5
- **throttle_increase_ms** – The rate at which Athena decreases the call delay. The default is 10 milliseconds.

**Databases and tables**

The Athena CloudWatch Metrics connector maps your namespaces, dimensions, metrics, and metric values into two tables in a single schema called default.

**The metrics table**

The metrics table contains the available metrics as uniquely defined by a combination of namespace, set, and name. The metrics table contains the following columns.

- **namespace** – A VARCHAR containing the namespace.
- **metric_name** – A VARCHAR containing the metric name.
- **dimensions** – A LIST of STRUCT objects composed of dim_name (VARCHAR) and dim_value (VARCHAR).
- **statistic** – A LIST of VARCH statistics (for example, p90, AVERAGE, ...) available for the metric.

**The metric_samples table**

The metric_samples table contains the available metric samples for each metric in the metrics table. The metric_samples table contains the following columns.

- **namespace** – A VARCHAR that contains the namespace.
- **metric_name** – A VARCHAR that contains the metric name.
- **dimensions** – A LIST of STRUCT objects composed of dim_name (VARCHAR) and dim_value (VARCHAR).
- **dim_name** – A VARCHAR convenience field that you can use to easily filter on a single dimension name.
- **dim_value** – A VARCHAR convenience field that you can use to easily filter on a single dimension value.
- **period** – An INT field that represents the "period" of the metric in seconds (for example, a 60 second metric).
- **timestamp** – A BIGINT field that represents the epoch time in seconds that the metric sample is for.
- **value** – A FLOAT8 field that contains the value of the sample.
- **statistic** – A VARCHAR that contains the statistic type of the sample (for example, AVERAGE or p90).

**Required Permissions**

Review the Policies section of the `athena-cloudwatch-metrics.yaml` file for full details on the IAM policies that this connector requires. The following is a brief summary.

- **Amazon S3 write access** – To successfully handle large queries, the connector requires write access to a location in Amazon S3.
• **Athena GetQueryExecution** – The connector uses this permission to fast-fail when the upstream Athena query has terminated.

• **CloudWatch Metrics ReadOnly** – The connector uses this permission to query your metrics data.

• **CloudWatch Logs Write** – The connector uses this access to write its diagnostic logs.

**Performance**

The Athena CloudWatch Metrics connector attempts to optimize queries against CloudWatch Metrics by parallelizing scans of the log streams required for your query. For certain time period, metric, namespace, and dimension filters, predicate pushdown is performed both within the Lambda function and within CloudWatch Logs.

**License information**

The Amazon Athena CloudWatch Metrics connector project is licensed under the **Apache-2.0 License**.

**See also**

For additional information about this connector, visit [the corresponding site](https://github.com) on GitHub.com.

**Amazon Athena AWS CMDB connector**

The Amazon Athena AWS CMDB connector enables Athena to communicate with various AWS services so that you can query them with SQL.

**Prerequisites**

- Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see [Deploying a connector and connecting to a data source](p. 62) or [Using the AWS Serverless Application Repository to deploy a data source connector](p. 64).

**Parameters**

Use the Lambda environment variables in this section to configure the AWS CMDB connector.

- **spill_bucket** – Specifies the Amazon S3 bucket for data that exceeds Lambda function limits.

- **spill_prefix** – (Optional) Defaults to a subfolder in the specified spill_bucket called athena-federation-spill. We recommend that you configure an Amazon S3 storage lifecycle on this location to delete spills older than a predetermined number of days or hours.

- **spill_put_request_headers** – (Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, "x-amz-server-side-encryption" : "AES256"). For other possible headers, see [PutObject](https://docs.aws.amazon.com/) in the Amazon Simple Storage Service API Reference.

- **kms_key_id** – (Optional) By default, any data that is spilled to Amazon S3 is encrypted using the AES-GCM authenticated encryption mode and a randomly generated key. To have your Lambda function use stronger encryption keys generated by KMS like a7e63k4b-8loc-40db-a2a1-4d0en2cd8331, you can specify a KMS key ID.

- **disable_spill_encryption** – (Optional) When set to `True`, disables spill encryption. Defaults to `False` so that data that is spilled to S3 is encrypted using AES-GCM – either using a randomly generated key or KMS to generate keys. Disabling spill encryption can improve performance, especially if your spill location uses [server-side encryption](https://docs.aws.amazon.com/).
• **default_ec2_image_owner** – (Optional) When set, controls the default Amazon EC2 image owner that filters *Amazon Machine Images (AMI)*. If you do not set this value and your query against the EC2 images table does not include a filter for owner, your results will include all public images.

### Databases and tables

The Athena AWS CMDB connector makes the following databases and tables available for querying your AWS resource inventory. For more information on the columns available in each table, run a `DESCRIBE database.table` statement using the Athena console or API.

- **ec2** – This database contains Amazon EC2 related resources, including the following.
  - **ebs_volumes** – Contains details of your Amazon EBS volumes.
  - **ec2_instances** – Contains details of your EC2 Instances.
  - **ec2_images** – Contains details of your EC2 Instance images.
  - **routing_tables** – Contains details of your VPC Routing Tables.
  - **security_groups** – Contains details of your security groups.
  - **subnets** – Contains details of your VPC Subnets.
  - **vpcs** – Contains details of your VPCs.

- **emr** – This database contains Amazon EMR related resources, including the following.
  - **emr_clusters** – Contains details of your EMR Clusters.

- **rds** – This database contains Amazon RDS related resources, including the following.
  - **rds_instances** – Contains details of your RDS Instances.

- **s3** – This database contains RDS related resources, including the following.
  - **buckets** – Contains details of your Amazon S3 buckets.
  - **objects** – Contains details of your Amazon S3 objects, excluding their contents.

### Required Permissions

Review the Policies section of the `athena-aws-cmdb.yaml` file for full details on the IAM policies that this connector requires. The following is a brief summary.

- **Amazon S3 write access** – To successfully handle large queries, the connector requires write access to a location in Amazon S3.
- **Athena GetQueryExecution** – The connector uses this permission to fast-fail when the upstream Athena query has terminated.
- **S3 List** – The connector uses this permission to list your Amazon S3 buckets and objects.
- **EC2 Describe** – The connector uses this permission to describe resources such as your Amazon EC2 instances, security groups, VPCs, and Amazon EBS volumes.
- **EMR Describe / List** – The connector uses this permission to describe your EMR clusters.
- **RDS Describe** – The connector uses this permission to describe your RDS Instances.
Performance

Currently, the Athena AWS CMDB connector does not support parallel scans. Predicate pushdown is performed within the Lambda function. Where possible, partial predicates are pushed to the services being queried. For example, a query for the details of a specific Amazon EC2 instance calls the EC2 API with the specific instance ID to run a targeted describe operation.

License information

The Amazon Athena AWS CMDB connector project is licensed under the Apache-2.0 License.

See also

For additional information about this connector, visit the corresponding site on GitHub.com.

Amazon Athena IBM Db2 connector

The Amazon Athena connector for Db2 enables Amazon Athena to run SQL queries on your IBM Db2 databases using JDBC.

Prerequisites

- Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).
- Set up a VPC and a security group before you use this connector. For more information, see Creating a VPC for a data source connector (p. 66).

Limitations

- Write DDL operations are not supported.
- In a multiplexer setup, the spill bucket and prefix are shared across all database instances.
- Any relevant Lambda limits. For more information, see Lambda quotas in the AWS Lambda Developer Guide.
- Date and timestamp data types in filter conditions must be cast to appropriate data types.

Terms

The following terms relate to the Db2 connector.

- **Database instance** – Any instance of a database deployed on premises, on Amazon EC2, or on Amazon RDS.
- **Handler** – A Lambda handler that accesses your database instance. A handler can be for metadata or for data records.
- **Metadata handler** – A Lambda handler that retrieves metadata from your database instance.
- **Record handler** – A Lambda handler that retrieves data records from your database instance.
- **Composite handler** – A Lambda handler that retrieves both metadata and data records from your database instance.
- **Property or parameter** – A database property used by handlers to extract database information. You configure these properties as Lambda environment variables.
- **Connection String** – A string of text used to establish a connection to a database instance.
- **Catalog** – A non-AWS Glue catalog registered with Athena that is a required prefix for the connection_string property.
• **Multiplexing handler** – A Lambda handler that can accept and use multiple database connections.

**Parameters**

Use the Lambda environment variables in this section to configure the Db2 connector.

**Connection string**

Use a JDBC connection string in the following format to connect to a database instance.

```
dbtwo://${jdbc_connection_string}
```

**Using a multiplexing handler**

You can use a multiplexer to connect to multiple database instances with a single Lambda function. Requests are routed by catalog name. Use the following classes in Lambda.

<table>
<thead>
<tr>
<th>Handler</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>Db2MuxCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>Db2MuxMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>Db2MuxRecordHandler</td>
</tr>
</tbody>
</table>

**Multiplexing handler parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$catalog_connection_string</td>
<td>Required. A database instance connection string. Prefix the string with the name of the catalog used in Athena. For example, if the catalog registered with Athena is mydbtwocatalog, then the environment variable name is mydbtwocatalog_connection_string.</td>
</tr>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when the catalog is lambda:${AWS_LAMBDA_FUNCTION_NAME}.</td>
</tr>
</tbody>
</table>

The following example properties are for a Db2 MUX Lambda function that supports two database instances: dbtwo1 (the default), and dbtwo2.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>dttwo://jdbc:db2://dttwo1.hostname:port/database_name:$secret1_name</td>
</tr>
<tr>
<td>dttwo_catalog1_connection_string</td>
<td>dttwo://jdbc:db2://dttwo1.hostname:port/database_name:$secret1_name</td>
</tr>
<tr>
<td>dttwo_catalog2_connection_string</td>
<td>dttwo://jdbc:db2://dttwo2.hostname:port/database_name:$secret2_name</td>
</tr>
</tbody>
</table>
Providing credentials

To provide a user name and password for your database in your JDBC connection string, you can use connection string properties or AWS Secrets Manager.

- **Connection String** – A user name and password can be specified as properties in the JDBC connection string.

- **AWS Secrets Manager** – To use the Athena Federated Query feature with AWS Secrets Manager, the VPC connected to your Lambda function should have internet access or a VPC endpoint to connect to Secrets Manager.

You can put the name of a secret in AWS Secrets Manager in your JDBC connection string. The connector replaces the secret name with the username and password values from Secrets Manager.

For Amazon RDS database instances, this support is tightly integrated. If you use Amazon RDS, we highly recommend using AWS Secrets Manager and credential rotation. If your database does not use Amazon RDS, store the credentials as JSON in the following format:

```
{"username": "${username}", "password": "${password}"}
```

**Example connection string with secret name**

The following string has the secret name ${secret_name}.

```
dbtwo://jdbc:db2://hostname:port/database_name:${secret_name}
```

The connector uses the secret name to retrieve secrets and provide the user name and password, as in the following example.

```
dbtwo://jdbc:db2://hostname:port/database_name:user=user_name;password=password;
```

**Using a single connection handler**

You can use the following single connection metadata and record handlers to connect to a single Db2 instance.

<table>
<thead>
<tr>
<th>Handler type</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>Db2CompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>Db2MetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>Db2RecordHandler</td>
</tr>
</tbody>
</table>

**Single connection handler parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when a catalog is not recognized.</td>
</tr>
</tbody>
</table>

The single connection handlers support one database instance and must provide a default connection string parameter. All other connection strings are ignored.
The following example property is for a single Db2 instance supported by a Lambda function.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>dbtwo://jdbc:db2://hostname:port/database_name:${secret_name}</td>
</tr>
</tbody>
</table>

**Spill parameters**

The Lambda SDK can spill data to Amazon S3. All database instances accessed by the same Lambda function spill to the same location.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spill_bucket</td>
<td>Required. Spill bucket name.</td>
</tr>
<tr>
<td>spill_prefix</td>
<td>Required. Spill bucket key prefix.</td>
</tr>
<tr>
<td>spill_put_request_headers</td>
<td>(Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, {&quot;x-amz-server-side-encryption&quot;: &quot;AES256&quot;}). For other possible headers, see PutObject in the Amazon Simple Storage Service API Reference.</td>
</tr>
</tbody>
</table>

**Data type support**

The following table shows the corresponding data types for JDBC and Arrow.

<table>
<thead>
<tr>
<th>Db2</th>
<th>Arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>DATE</td>
<td>DATEDAY</td>
</tr>
<tr>
<td>TIME</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>DATEGILL</td>
</tr>
<tr>
<td>DATETIME</td>
<td>DATEGILL</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>BOOL</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INT</td>
</tr>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>REAL</td>
<td>FLOAT8</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>FLOAT8</td>
</tr>
<tr>
<td>DECFLOAT</td>
<td>FLOAT8</td>
</tr>
</tbody>
</table>
Partitions and splits

A partition is represented by one or more partition columns of type `varchar`. The Db2 connector creates partitions using the following organization schemes.

- Distribute by hash
- Partition by range
- Organize by dimensions

The connector retrieves partition details such as the number of partitions and column name from one or more Db2 metadata tables. Splits are created based upon the number of partitions identified.

Best practices

As a best practice, use predicate pushdown when you query Athena, as in the following examples.

```sql
SELECT *
FROM "lambda:LAMBDA_NAME"."SCHEMA_NAME"."TABLE_NAME"
WHERE integercol = 2147483647
```

```sql
SELECT *
FROM "lambda:LAMBDA_NAME"."SCHEMA_NAME"."TABLE_NAME"
WHERE timestampcol >= TIMESTAMP '2018-03-25 07:30:58.878'
```

License information

By using this connector, you acknowledge the inclusion of third party components, a list of which can be found in the `pom.xml` file for this connector, and agree to the terms in the respective third party licenses provided in the `LICENSE.txt` file on GitHub.com.

See also

For the latest JDBC driver version information, see the `pom.xml` file for the Db2 connector on GitHub.com.

For additional information about this connector, visit the corresponding site on GitHub.com.

Amazon Athena DocumentDB connector

The Amazon Athena DocumentDB connector enables Athena to communicate with your DocumentDB instances so that you can query your DocumentDB data with SQL. The connector also works with any endpoint that is compatible with MongoDB.

Unlike traditional relational data stores, Amazon DocumentDB collections do not have set schema. DocumentDB does not have a metadata store. Each entry in a DocumentDB collection can have different fields and data types.

The DocumentDB connector supports two mechanisms for generating table schema information: basic schema inference and AWS Glue Data Catalog metadata.

Schema inference is the default. This option scans a small number of documents in your collection, forms a union of all fields, and coerces fields that have non-overlapping data types. This option works well for collections that have mostly uniform entries.

For collections with a greater variety of data types, the connector supports retrieving metadata from the AWS Glue Data Catalog. If the connector sees a AWS Glue database and table that match your DocumentDB database and collection names, it gets its schema information from the corresponding
AWS Glue table. When you create your AWS Glue table, we recommend that you make it a superset of all fields that you might want to access from your DocumentDB collection.

Prerequisites

- Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).

Parameters

Use the Lambda environment variables in this section to configure the DocumentDB connector.

- **spill_bucket** – Specifies the Amazon S3 bucket for data that exceeds Lambda function limits.
- **spill_prefix** – (Optional) Defaults to a subfolder in the specified spill_bucket called athena-federation-spill. We recommend that you configure an Amazon S3 storage lifecycle on this location to delete spills older than a predetermined number of days or hours.
- **spill_put_request_headers** – (Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, {"x-amz-server-side-encryption" : "AES256"}). For other possible headers, see PutObject in the Amazon Simple Storage Service API Reference.
- **kms_key_id** – (Optional) By default, any data that is spilled to Amazon S3 is encrypted using the AES-GCM authenticated encryption mode and a randomly generated key. To have your Lambda function use stronger encryption keys generated by KMS like a7e63k4b-8loc-40db-a2a1-4d0en2cd8331, you can specify a KMS key ID.
- **disable_spill_encryption** – (Optional) When set to True, disables spill encryption. Defaults to False so that data that is spilled to S3 is encrypted using AES-GCM – either using a randomly generated key or KMS to generate keys. Disabling spill encryption can improve performance, especially if your spill location uses server-side encryption.
- **disable_glue** – (Optional) If present and set to true, the connector does not attempt to retrieve supplemental metadata from AWS Glue.
- **glue_catalog** – (Optional) Use this option to specify a cross-account AWS Glue catalog (p. 24). By default, the connector attempts to get metadata from its own AWS Glue account.
- **default_docdb** – If present, specifies a DocumentDB connection string to use when no catalog-specific environment variable exists.
- **disable_projection_and_casing** – (Optional) Disables projection and casing. Use if you want to query Amazon DocumentDB tables that use case sensitive column names. The disable_projection_and_casing parameter uses the following values to specify the behavior of casing and column mapping:
  - **false** – This is the default setting. Projection is enabled, and the connector expects all column names to be in lower case.
  - **true** – Disables projection and casing. When using the disable_projection_and_casing parameter, keep in mind the following points:
    - Use of the parameter can result in higher bandwidth usage if your Lambda function is not in the same AWS Region as your Amazon DocumentDB instances.
    - Because a larger number of bytes is transferred and because the larger number of bytes requires a higher deserialization time, overall latency can increase.

Specifying connection strings

You can provide one or more properties that define the DocumentDB connection details for the DocumentDB instances that you use with the connector. To do this, set a Lambda environment variable
that corresponds to the catalog name that you want to use in Athena. For example, suppose you want to use the following queries to query two different DocumentDB instances from Athena:

```
SELECT * FROM "docdb_instance_1".database.table
```

```
SELECT * FROM "docdb_instance_2".database.table
```

Before you can use these two SQL statements, you must add two environment variables to your Lambda function: docdb_instance_1 and docdb_instance_2. The value for each should be a DocumentDB connection string in the following format:

```
mongodb://:@:/?ssl=true&ssl_ca_certs=rds-combined-ca-bundle.pem&replicaSet=rs0
```

**Using secrets**

You can optionally use AWS Secrets Manager for part or all of the value for your connection string details. To use the Athena Federated Query feature with Secrets Manager, the VPC connected to your Lambda function should have internet access or a VPC endpoint to connect to Secrets Manager.

If you use the syntax ${my_secret} to put the name of a secret from Secrets Manager in your connection string, the connector replaces the secret name with your user name and password values from Secrets Manager.

For example, suppose you set the Lambda environment variable for docdb_instance_1 to the following value:

```
mongodb://${docdb_instance_1_creds}@myhostname.com:123/?ssl=true&ssl_ca_certs=rds-combined-ca-bundle.pem&replicaSet=rs0
```

The Athena Query Federation SDK automatically attempts to retrieve a secret named docdb_instance_1_creds from Secrets Manager and inject that value in place of ${docdb_instance_1_creds}. Any part of the connection string that is enclosed by the ${ } character combination is interpreted as a secret from Secrets Manager. If you specify a secret name that the connector cannot find in Secrets Manager, the connector does not replace the text.

**Setting up databases and tables in AWS Glue**

Because the connector's built-in schema inference capability scans a limited number of documents and supports only a subset of data types, you might want to use AWS Glue for metadata instead.

To enable an AWS Glue table for use with Amazon DocumentDB, you must have a AWS Glue database and table for the DocumentDB database and collection that you want to supply supplemental metadata for.

**To use an AWS Glue table for supplemental metadata**

1. When you edit the table and database in the AWS Glue console, add the following table property.

   - **docdb-metadata-flag** – This property indicates to the DocumentDB connector that the connector can use the table for supplemental metadata. You can provide any value for docdb-metadata-flag as long as the docdb-metadata-flag property is present in the list of table properties.

2. (Optional) Add the **sourceTable** table property. This property defines the source table name in Amazon DocumentDB. Use this property if AWS Glue table naming rules prevent you from creating an AWS Glue table with the same name as your Amazon DocumentDB table. For example, capital letters are not permitted in AWS Glue table names, but they are permitted in Amazon DocumentDB table names.
3. Make sure that you use the data types appropriate for AWS Glue as listed in this document.

Data type support
This section lists the data types that the DocumentDB connector uses for schema inference, and the data types when AWS Glue metadata is used.

Schema inference data types
The schema inference feature of the DocumentDB connector attempts to infer values as belonging to one of the following data types. The table shows the corresponding data types for Amazon DocumentDB, Java, and Apache Arrow.

<table>
<thead>
<tr>
<th>Apache Arrow</th>
<th>Java or DocDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR</td>
<td>String</td>
</tr>
<tr>
<td>INT</td>
<td>Integer</td>
</tr>
<tr>
<td>BIGINT</td>
<td>Long</td>
</tr>
<tr>
<td>BIT</td>
<td>Boolean</td>
</tr>
<tr>
<td>FLOAT4</td>
<td>Float</td>
</tr>
<tr>
<td>FLOAT8</td>
<td>Double</td>
</tr>
<tr>
<td>TIMESTAMPSEC</td>
<td>Date</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>ObjectId</td>
</tr>
<tr>
<td>LIST</td>
<td>List</td>
</tr>
<tr>
<td>STRUCT</td>
<td>Document</td>
</tr>
</tbody>
</table>

AWS Glue data types
If you use AWS Glue for supplemental metadata, you can configure the following data types. The table shows the corresponding data types for AWS Glue and Apache Arrow.

<table>
<thead>
<tr>
<th>AWS Glue</th>
<th>Apache Arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>INT</td>
</tr>
<tr>
<td>bigint</td>
<td>BIGINT</td>
</tr>
<tr>
<td>double</td>
<td>FLOAT8</td>
</tr>
<tr>
<td>float</td>
<td>FLOAT4</td>
</tr>
<tr>
<td>boolean</td>
<td>BIT</td>
</tr>
<tr>
<td>binary</td>
<td>VARBINARY</td>
</tr>
<tr>
<td>string</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>List</td>
<td>LIST</td>
</tr>
<tr>
<td>Struct</td>
<td>STRUCT</td>
</tr>
</tbody>
</table>
Required Permissions

Review the Policies section of the athena-docdb.yaml file for full details on the IAM policies that this connector requires. The following is a brief summary.

- **Amazon S3 write access** – To successfully handle large queries, the connector requires write access to a location in Amazon S3.
- **Athena GetQueryExecution** – The connector uses this permission to fast-fail when the upstream Athena query has terminated.
- **AWS Glue Data Catalog** – The DocumentDB connector requires read only access to the AWS Glue Data Catalog to obtain schema information.
- **CloudWatch Logs** – The connector requires access to CloudWatch Logs for storing logs.
- **AWS Secrets Manager read access** – If you choose to store DocumentDB endpoint details in Secrets Manager, you must grant the connector access to those secrets.
- **VPC access** – The connector requires the ability to attach and detach interfaces to your VPC so that it can connect to it and communicate with your DocumentDB instances.

Performance

The Athena Amazon DocumentDB connector attempts to push down predicates as part of its queries, but it does not currently support parallel scans.

See also

For additional information about this connector, visit the corresponding site on GitHub.com.

Amazon Athena DynamoDB connector

The Amazon Athena DynamoDB connector enables Amazon Athena to communicate with DynamoDB so that you can query your tables with SQL. Write operations like `INSERT INTO` (p. 697) are not supported.

Prerequisites

- Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).

Parameters

Use the Lambda environment variables in this section to configure the DynamoDB connector.

- **spill_bucket** – Specifies the Amazon S3 bucket for data that exceeds Lambda function limits.
- **spill_prefix** – (Optional) Defaults to a subfolder in the specified spill_bucket called athena-federation-spill. We recommend that you configure an Amazon S3 storage lifecycle on this location to delete spills older than a predetermined number of days or hours.
- **spill_put_request_headers** – (Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, `{"x-amz-server-side-encryption" : "AES256"}`). For other possible headers, see PutObject in the Amazon Simple Storage Service API Reference.
- **kms_key_id** – (Optional) By default, any data that is spilled to Amazon S3 is encrypted using the AES-GCM authenticated encryption mode and a randomly generated key. To have your Lambda function
use stronger encryption keys generated by KMS like a7e63k4b-8loc-40db-a2a1-4d0en2cd8331, you can specify a KMS key ID.

- **disable_spill_encryption** – (Optional) When set to True, disables spill encryption. Defaults to False so that data that is spilled to S3 is encrypted using AES-GCM – either using a randomly generated key or KMS to generate keys. Disabling spill encryption can improve performance, especially if your spill location uses server-side encryption.

- **disable_glue** – (Optional) If present and set to true, the connector does not attempt to retrieve supplemental metadata from AWS Glue.

- **glue_catalog** – (Optional) Use this option to specify a cross-account AWS Glue catalog (p. 24). By default, the connector attempts to get metadata from its own AWS Glue account.

- **disable_projection_and_casing** – (Optional) Disables projection and casing. Use if you want to query DynamoDB tables that have casing in their column names and you do not want to specify a columnMapping property on your AWS Glue table.

The disable_projection_and_casing parameter uses the following values to specify the behavior of casing and column mapping:

- **auto** – Disables projection and casing when a previously unsupported type is detected and column name mapping is not set on the table. This is the default setting.

- **always** – Disables projection and casing unconditionally. This is useful when you have casing in your DynamoDB column names but do not want to specify any column name mapping.

When using the disable_projection_and_casing parameter, keep in mind the following points:

- Use of the parameter can result in higher bandwidth usage if your Lambda function is not in the same region as your DynamoDB table.

- Because a larger number of bytes is transferred and because the larger number of bytes requires a higher deserialization time, overall latency can increase.

### Setting up databases and tables in AWS Glue

Because the connector's built-in schema inference capability is limited, you might want to use AWS Glue for metadata. To enable an AWS Glue table for use with DynamoDB, you must have an AWS Glue table for the DynamoDB table that you want to supply supplemental metadata for.

#### To use an AWS Glue table for supplemental metadata

1. When you edit the table and database in the AWS Glue console, add the following table properties as required. If you use the AWS Glue DynamoDB crawler, these properties are automatically set.

   - **dynamodb** – String that indicates to the Athena DynamoDB connector that the table can be used for supplemental metadata. You can enter the dynamodb string in any one of the following places:
     - In the table properties under a field called classification (exact match).
     - In the table storage descriptor location field (sub-string match).
     - In the table storage descriptor parameters under a field called classification (exact match).

   - **dynamo-db-flag** – String that indicates that the database contains tables that the Athena DynamoDB connector is using for supplemental metadata. This is required for AWS Glue databases other than default. The dynamo-db-flag property is useful for filtering out irrelevant databases in accounts that have many of them. This string should be in the Location URI of the AWS Glue database (sub-string match).

   - **sourceTable** – Optional table property that defines the source table name in DynamoDB. Use this if AWS Glue table naming rules prevent you from creating a AWS Glue table with the same name as your DynamoDB table. For example, capital letters are not permitted in AWS Glue table names, but they are permitted in DynamoDB table names.

   - **columnMapping** – Optional table property that defines column name mappings. Use this if AWS Glue column naming rules prevent you from creating a AWS Glue table with the same column
names as your DynamoDB table. For example, capital letters are not permitted in AWS Glue column names but are permitted in DynamoDB column names. The property value is expected to be in the format col1=Col1,col2=Col2. Note that column mapping applies only to top level column names and not to nested fields.

- **defaultTimeZone** – Optional table property that is applied to date or datetime values that do not have an explicit time zone. Setting this value is a good practice to avoid discrepancies between the data source default time zone and the Athena session time zone.

- **datetimeFormatMapping** – Optional table property that specifies the date or datetime format to use when parsing data from a column of the AWS Glue date or timestamp data type. If this property is not specified, the connector attempts to infer an ISO-8601 format. If the connector cannot infer the date or datetime format or parse the raw string, then the value is omitted from the result.

The datetimeFormatMapping value should be in the format col1=someformat1,col2=someformat2. Following are some example formats:

```
yyyyMMdd'T'HHmmss
ddMMyyyy'T'HH:mm:ss
```

If your column has date or datetime values without a time zone and you want to use the column in the WHERE clause, set the datetimeFormatMapping property for the column.

2. If you define your columns manually, make sure that you use the appropriate data types. If you used a crawler, validate the columns and types that the crawler discovered.

**Required Permissions**

Review the Policies section of the `athena-dynamodb.yaml` file for full details on the IAM policies that this connector requires. The following is a brief summary.

- **Amazon S3 write access** – To successfully handle large queries, the connector requires write access to a location in Amazon S3.

- **Athena GetQueryExecution** – The connector uses this permission to fast-fail when the upstream Athena query has terminated.

- **AWS Glue Data Catalog** – The DynamoDB connector requires read only access to the AWS Glue Data Catalog to obtain schema information.

- **CloudWatch Logs** – The connector requires access to CloudWatch Logs for storing logs.

- **DynamoDB read access** – The connector uses the DescribeTable, ListSchemas, ListTables, Query, and Scan API operations.

**Performance**

The Athena DynamoDB connector supports parallel scans and attempts to push down predicates as part of its DynamoDB queries. A hash key predicate with $X$ distinct values results in $X$ Query calls to DynamoDB. All other predicate scenarios result in $Y$ number of scan calls, where $Y$ is heuristically determined based on the size of your table and its provisioned throughput.

Predicate pushdown is performed within the Lambda function to decrease the data scanned by the query. However, selecting a subset of columns sometimes results in longer query execution runtime. LIMIT clauses reduce the amount of data scanned, but if a predicate is not provided, you should expect SELECT queries with a LIMIT clause to scan at least 16mb of data.

The DynamoDB connector scans more data for larger datasets than for smaller datasets regardless of the LIMIT clause applied. For example, the query `SELECT * LIMIT 10000` scans more data for a larger underlying dataset than a smaller underlying dataset.
Costs

The costs for use of the connector depends on the underlying AWS resources that are used. Because queries that use scans can consume a large number of read capacity units (RCUs), consider the information for Amazon DynamoDB pricing carefully.

See also

For additional information about this connector, visit the corresponding site on GitHub.com.

Amazon Athena HBase connector

The Amazon Athena HBase connector enables Amazon Athena to communicate with your Apache HBase instances so that you can query your HBase data with SQL.

Unlike traditional relational data stores, HBase collections do not have set schema. HBase does not have a metadata store. Each entry in a HBase collection can have different fields and data types.

The HBase connector supports two mechanisms for generating table schema information: basic schema inference and AWS Glue Data Catalog metadata.

Schema inference is the default. This option scans a small number of documents in your collection, forms a union of all fields, and coerces fields that have non overlapping data types. This option works well for collections that have mostly uniform entries.

For collections with a greater variety of data types, the connector supports retrieving metadata from the AWS Glue Data Catalog. If the connector sees an AWS Glue database and table that match your HBase namespace and collection names, it gets its schema information from the corresponding AWS Glue table. When you create your AWS Glue table, we recommend that you make it a superset of all fields that you might want to access from your HBase collection.

Prerequisites

- Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).

Parameters

Use the Lambda environment variables in this section to configure the HBase connector.

- **spill_bucket** – Specifies the Amazon S3 bucket for data that exceeds Lambda function limits.
- **spill_prefix** – (Optional) Defaults to a subfolder in the specified spill_bucket called athena-federation-spill. We recommend that you configure an Amazon S3 storage lifecycle on this location to delete spills older than a predetermined number of days or hours.
- **spill_put_request_headers** – (Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, \{"x-amz-server-side-encryption" : "AES256"\}). For other possible headers, see PutObject in the Amazon Simple Storage Service API Reference.
- **kms_key_id** – (Optional) By default, any data that is spilled to Amazon S3 is encrypted using the AES-GCM authenticated encryption mode and a randomly generated key. To have your Lambda function use stronger encryption keys generated by KMS like a7e63k4b-8loc-40db-a2a1-4d0en2cd8331, you can specify a KMS key ID.
• **disable_spill_encryption** – (Optional) When set to True, disables spill encryption. Defaults to False so that data that is spilled to S3 is encrypted using AES-GCM – either using a randomly generated key or KMS to generate keys. Disabling spill encryption can improve performance, especially if your spill location uses server-side encryption.

• **disable_glue** – (Optional) If present and set to true, the connector does not attempt to retrieve supplemental metadata from AWS Glue.

• **glue_catalog** – (Optional) Use this option to specify a cross-account AWS Glue catalog (p. 24). By default, the connector attempts to get metadata from its own AWS Glue account.

• **default_hbase** – If present, specifies an HBase connection string to use when no catalog-specific environment variable exists.

### Specifying connection strings

You can provide one or more properties that define the HBase connection details for the HBase instances that you use with the connector. To do this, set a Lambda environment variable that corresponds to the catalog name that you want to use in Athena. For example, suppose you want to use the following queries to query two different HBase instances from Athena:

```
SELECT * FROM "hbase_instance_1".database.table
```

```
SELECT * FROM "hbase_instance_2".database.table
```

Before you can use these two SQL statements, you must add two environment variables to your Lambda function: `hbase_instance_1` and `hbase_instance_2`. The value for each should be a HBase connection string in the following format:

```
master_hostname:hbase_port:zookeeper_port
```

### Using secrets

You can optionally use AWS Secrets Manager for part or all of the value for your connection string details. To use the Athena Federated Query feature with Secrets Manager, the VPC connected to your Lambda function should have internet access or a VPC endpoint to connect to Secrets Manager.

If you use the syntax `${my_secret}` to put the name of a secret from Secrets Manager in your connection string, the connector replaces the secret name with your user name and password values from Secrets Manager.

For example, suppose you set the Lambda environment variable for `hbase_instance_1` to the following value:

```
${hbase_host_1}:${hbase_master_port_1}:${hbase_zookeeper_port_1}
```

The Athena Query Federation SDK automatically attempts to retrieve a secret named `hbase_instance_1_creds` from Secrets Manager and inject that value in place of `${hbase_instance_1_creds}`. Any part of the connection string that is enclosed by the `${ }` character combination is interpreted as a secret from Secrets Manager. If you specify a secret name that the connector cannot find in Secrets Manager, the connector does not replace the text.

### Setting up databases and tables in AWS Glue

The connector’s built-in schema inference supports only values that are serialized in HBase as strings (for example, `String.valueOf(int)`). Because the connector’s built-in schema inference capability is
To use an AWS Glue table for supplemental metadata

1. When you edit the table and database in the AWS Glue console, add the following table properties:

   - **hbase-metadata-flag** – This property indicates to the HBase connector that the connector can use the table for supplemental metadata. You can provide any value for hbase-metadata-flag as long as the hbase-metadata-flag property is present in the list of table properties.
   - **hbase-native-storage-flag** – Use this flag to toggle the two value serialization modes supported by the connector. By default, when this field is not present, the connector assumes all values are stored in HBase as strings. As such it will attempt to parse data types such as INT, BIGINT, and DOUBLE from HBase as strings. If this field is set with any value on the table in AWS Glue, the connector switches to "native" storage mode and attempts to read INT, BIGINT, BIT, and DOUBLE as bytes by using the following functions:

     ```java
     ByteBuffer.wrap(value).getInt()
     ByteBuffer.wrap(value).getLong()
     ByteBuffer.wrap(value).get()
     ByteBuffer.wrap(value).getDouble()
     ```

2. Make sure that you use the data types appropriate for AWS Glue as listed in this document.

Modeling column families

The Athena HBase connector supports two ways to model HBase column families: fully qualified (flattened) naming like `family:column`, or using STRUCT objects.

In the STRUCT model, the name of the STRUCT field should match the column family, and children of the STRUCT should match the names of the columns of the family. However, because predicate push down and columnar reads are not yet fully supported for complex types like STRUCT, using STRUCT is currently not advised.

The following image shows a table configured in AWS Glue that uses a combination of the two approaches.
Data type support

The connector retrieves all HBase values as the basic byte type. Then, based on how you defined your tables in AWS Glue Data Catalog, it maps the values into one of the Apache Arrow data types in the following table.

<table>
<thead>
<tr>
<th>AWS Glue data type</th>
<th>Apache Arrow data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>INT</td>
</tr>
<tr>
<td>bigint</td>
<td>BIGINT</td>
</tr>
<tr>
<td>double</td>
<td>FLOAT8</td>
</tr>
</tbody>
</table>
### Required Permissions

Review the Policies section of the `athena-hbase.yaml` file for full details on the IAM policies that this connector requires. The following is a brief summary.

- **Amazon S3 write access** – To successfully handle large queries, the connector requires write access to a location in Amazon S3.
- **Athena GetQueryExecution** – The connector uses this permission to fast-fail when the upstream Athena query has terminated.
- **AWS Glue Data Catalog** – The HBase connector requires read only access to the AWS Glue Data Catalog to obtain schema information.
- **CloudWatch Logs** – The connector requires access to CloudWatch Logs for storing logs.
- **AWS Secrets Manager read access** – If you choose to store HBase endpoint details in Secrets Manager, you must grant the connector access to those secrets.
- **VPC access** – The connector requires the ability to attach and detach interfaces to your VPC so that it can connect to it and communicate with your HBase instances.

### Performance

The Athena HBase connector attempts to parallelize queries against your HBase instance by reading each region server in parallel. The connector performs predicate pushdown within the Lambda function and, where possible, uses filters to push down into HBase.

### License information

The Amazon Athena HBase connector project is licensed under the [Apache-2.0 License](https://www.apache.org/licenses/LICENSE-2|0.html).

### See also

For additional information about this connector, visit the corresponding site on GitHub.com.

### Amazon Athena Hortonworks connector

The Amazon Athena connector for Hortonworks enables Amazon Athena to run SQL queries on the Cloudera Hortonworks data platform. The connector transforms your Athena SQL queries to their equivalent HiveQL syntax.

### Prerequisites

- Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a
data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).

- Set up a VPC and a security group before you use this connector. For more information, see Creating a VPC for a data source connector (p. 66).

## Limitations

- Write DDL operations are not supported.
- In a multiplexer setup, the spill bucket and prefix are shared across all database instances.
- Any relevant Lambda limits. For more information, see Lambda quotas in the AWS Lambda Developer Guide.

## Terms

The following terms relate to the Hortonworks Hive connector.

- **Database instance** – Any instance of a database deployed on premises, on Amazon EC2, or on Amazon RDS.
- **Handler** – A Lambda handler that accesses your database instance. A handler can be for metadata or for data records.
- **Metadata handler** – A Lambda handler that retrieves metadata from your database instance.
- **Record handler** – A Lambda handler that retrieves data records from your database instance.
- **Composite handler** – A Lambda handler that retrieves both metadata and data records from your database instance.
- **Property or parameter** – A database property used by handlers to extract database information. You configure these properties as Lambda environment variables.
- **Connection String** – A string of text used to establish a connection to a database instance.
- **Catalog** – A non-AWS Glue catalog registered with Athena that is a required prefix for the connection_string property.
- **Multiplexing handler** – A Lambda handler that can accept and use multiple database connections.

## Parameters

Use the Lambda environment variables in this section to configure the Hortonworks Hive connector.

### Connection string

Use a JDBC connection string in the following format to connect to a database instance.

```
hive://${jdbc_connection_string}
```

### Using a multiplexing handler

You can use a multiplexer to connect to multiple database instances with a single Lambda function. Requests are routed by catalog name. Use the following classes in Lambda.

<table>
<thead>
<tr>
<th>Handler</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>HiveMuxCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>HiveMuxMetadataHandler</td>
</tr>
</tbody>
</table>
### Handler Class

| Record handler | HiveMuxRecordHandler |

### Multiplexing handler parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$catalog_connection_string</td>
<td>Required. A database instance connection string. Prefix the string with the name of the catalog used in Athena. For example, if the catalog registered with Athena is myhivecatalog, then the environment variable name is myhivecatalog_connection_string.</td>
</tr>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when the catalog is lambda:${AWS_LAMBDA_FUNCTION_NAME}.</td>
</tr>
</tbody>
</table>

The following example properties are for a Hive MUX Lambda function that supports two database instances: hive1 (the default), and hive2.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>hive://jdbc:hive2://hive1:10000/default?${Test/RDS/hive1}</td>
</tr>
<tr>
<td>hive_catalog1_connection_string</td>
<td>hive://jdbc:hive2://hive1:10000/default?${Test/RDS/hive1}</td>
</tr>
</tbody>
</table>

### Providing credentials

To provide a user name and password for your database in your JDBC connection string, you can use connection string properties or AWS Secrets Manager.

- **Connection String** – A user name and password can be specified as properties in the JDBC connection string.
- **AWS Secrets Manager** – To use the Athena Federated Query feature with AWS Secrets Manager, the VPC connected to your Lambda function should have internet access or a VPC endpoint to connect to Secrets Manager.

You can put the name of a secret in AWS Secrets Manager in your JDBC connection string. The connector replaces the secret name with the username and password values from Secrets Manager.

For Amazon RDS database instances, this support is tightly integrated. If you use Amazon RDS, we highly recommend using AWS Secrets Manager and credential rotation. If your database does not use Amazon RDS, store the credentials as JSON in the following format:

```
{"username": "${username}", "password": "${password}"}
```

### Example connection string with secret name

The following string has the secret name ${Test/RDS/hive1host}.
The connector uses the secret name to retrieve secrets and provide the user name and password, as in the following example.

```
hive://jdbc:hive2://hive1host:10000/default?...&UID=sample2&PWD=sample2&...
```

Currently, the Hortonworks Hive connector recognizes the UID and PWD JDBC properties.

**Using a single connection handler**

You can use the following single connection metadata and record handlers to connect to a single Hortonworks Hive instance.

<table>
<thead>
<tr>
<th>Handler type</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>HiveCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>HiveMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>HiveRecordHandler</td>
</tr>
</tbody>
</table>

**Single connection handler parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when a catalog is not recognized.</td>
</tr>
</tbody>
</table>

The single connection handlers support one database instance and must provide a default connection string parameter. All other connection strings are ignored.

The following example property is for a single Hortonworks Hive instance supported by a Lambda function.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>hive://jdbc:hive2://hive1host:10000/default?secret=${Test/RDS/hive1host}</td>
</tr>
</tbody>
</table>

**Spill parameters**

The Lambda SDK can spill data to Amazon S3. All database instances accessed by the same Lambda function spill to the same location.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spill_bucket</td>
<td>Required. Spill bucket name.</td>
</tr>
<tr>
<td>spill_prefix</td>
<td>Required. Spill bucket key prefix.</td>
</tr>
</tbody>
</table>
| spill_put_request_headers | (Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, 
"x-amz-server-side-encryption": |
Parameter | Description
--- | ---
"AES256" | For other possible headers, see `PutObject` in the Amazon Simple Storage Service API Reference.

Data type support

The following table shows the corresponding data types for JDBC, Hortonworks Hive, and Arrow.

<table>
<thead>
<tr>
<th>JDBC</th>
<th>Hortonworks Hive</th>
<th>Arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Boolean</td>
<td>Bit</td>
</tr>
<tr>
<td>Integer</td>
<td>TINYINT</td>
<td>Tiny</td>
</tr>
<tr>
<td>Short</td>
<td>SMALLINT</td>
<td>Smallint</td>
</tr>
<tr>
<td>Integer</td>
<td>INT</td>
<td>Int</td>
</tr>
<tr>
<td>Long</td>
<td>BIGINT</td>
<td>Bigint</td>
</tr>
<tr>
<td>float</td>
<td>float4</td>
<td>Float4</td>
</tr>
<tr>
<td>Double</td>
<td>float8</td>
<td>Float8</td>
</tr>
<tr>
<td>Date</td>
<td>date</td>
<td>DateDay</td>
</tr>
<tr>
<td>Timestamp</td>
<td>timestamp</td>
<td>DateMilli</td>
</tr>
<tr>
<td>String</td>
<td>VARCHAR</td>
<td>Varchar</td>
</tr>
<tr>
<td>Bytes</td>
<td>bytes</td>
<td>Varbinary</td>
</tr>
<tr>
<td>BigDecimal</td>
<td>Decimal</td>
<td>Decimal</td>
</tr>
<tr>
<td>ARRAY</td>
<td>N/A (see note)</td>
<td>List</td>
</tr>
</tbody>
</table>

Note

Currently, Hortonworks Hive does not support the aggregate types ARRAY, MAP, STRUCT, or UNIONTYPE. Columns of aggregate types are treated as VARCHAR columns in SQL.

Partitions and splits

Partitions are used to determine how to generate splits for the connector. Athena constructs a synthetic column of type VARCHAR that represents the partitioning scheme for the table to help the connector generate splits. The connector does not modify the actual table definition.

Performance tuning

Hortonworks Hive supports static partitions. The Athena Lambda connector can retrieve data from these partitions in parallel. If you want to query very large datasets with uniform partition distribution, static partitioning is highly recommended.

License information

By using this connector, you acknowledge the inclusion of third party components, a list of which can be found in the pom.xml file for this connector, and agree to the terms in the respective third party licenses provided in the LICENSE.txt file on GitHub.com.
See also

For the latest JDBC driver version information, see the pom.xml file for the Hortonworks Hive connector on GitHub.com.

For additional information about this connector, visit the corresponding site on GitHub.com.

Amazon Athena MySQL connector

The Amazon Athena Lambda MySQL connector enables Amazon Athena to access MySQL databases.

Prerequisites

• Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).

• Set up a VPC and a security group before you use this connector. For more information, see Creating a VPC for a data source connector (p. 66).

Limitations

• Write DDL operations are not supported.
• In a multiplexer setup, the spill bucket and prefix are shared across all database instances.
• Any relevant Lambda limits. For more information, see Lambda quotas in the AWS Lambda Developer Guide.
• Because Athena converts queries to lower case, MySQL table names must be in lower case. For example, Athena queries against a table named myTable will fail.

Terms

The following terms relate to the MySQL connector.

• Database instance – Any instance of a database deployed on premises, on Amazon EC2, or on Amazon RDS.
• Handler – A Lambda handler that accesses your database instance. A handler can be for metadata or for data records.
• Metadata handler – A Lambda handler that retrieves metadata from your database instance.
• Record handler – A Lambda handler that retrieves data records from your database instance.
• Composite handler – A Lambda handler that retrieves both metadata and data records from your database instance.
• Property or parameter – A database property used by handlers to extract database information. You configure these properties as Lambda environment variables.
• Connection String – A string of text used to establish a connection to a database instance.
• Catalog – A non-AWS Glue catalog registered with Athena that is a required prefix for the connection_string property.
• Multiplexing handler – A Lambda handler that can accept and use multiple database connections.

Parameters

Use the Lambda environment variables in this section to configure the MySQL connector.
Connection string

Use a JDBC connection string in the following format to connect to a database instance.

```
mysql://${jdbc_connection_string}
```

**Note**

If you receive the error `java.sql.SQLException: Zero date value prohibited when doing a SELECT query on a MySQL table`, add the following parameter to your connection string:

```
zeroDateTimeBehavior=convertToNull
```

For more information, see Error 'Zero date value prohibited' while trying to select from MySQL table on GitHub.com.

Using a multiplexing handler

You can use a multiplexer to connect to multiple database instances with a single Lambda function. Requests are routed by catalog name. Use the following classes in Lambda.

<table>
<thead>
<tr>
<th>Handler</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>MySqlMuxCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>MySqlMuxMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>MySqlMuxRecordHandler</td>
</tr>
</tbody>
</table>

Multiplexing handler parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$catalog_connection_string</td>
<td>Required. A database instance connection string. Prefix the string with the name of the catalog used in Athena. For example, if the catalog registered with Athena is <code>mymysqlcatalog</code>, then the environment variable name is <code>mymysqlcatalog_connection_string</code>.</td>
</tr>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when the catalog is <code>lambda:${AWS_LAMBDA_FUNCTION_NAME}</code>.</td>
</tr>
</tbody>
</table>

The following example properties are for a MySql MUX Lambda function that supports two database instances: `mysql1` (the default), and `mysql2`.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td><code>mysql://jdbc:mysql://mysql2.host:3333/default?user=sample2&amp;password=sample2</code></td>
</tr>
<tr>
<td>mysql_catalog1_connection_string</td>
<td><code>mysql://jdbc:mysql://mysql1.host:3306/default?${Test/RDS/MySql1}</code></td>
</tr>
<tr>
<td>mysql_catalog2_connection_string</td>
<td><code>mysql://jdbc:mysql://mysql2.host:3333/default?user=sample2&amp;password=sample2</code></td>
</tr>
</tbody>
</table>
Providing credentials

To provide a user name and password for your database in your JDBC connection string, you can use connection string properties or AWS Secrets Manager.

- **Connection String** – A user name and password can be specified as properties in the JDBC connection string.
- **AWS Secrets Manager** – To use the Athena Federated Query feature with AWS Secrets Manager, the VPC connected to your Lambda function should have internet access or a VPC endpoint to connect to Secrets Manager.

You can put the name of a secret in AWS Secrets Manager in your JDBC connection string. The connector replaces the secret name with the username and password values from Secrets Manager.

For Amazon RDS database instances, this support is tightly integrated. If you use Amazon RDS, we highly recommend using AWS Secrets Manager and credential rotation. If your database does not use Amazon RDS, store the credentials as JSON in the following format:

```json
{"username": "${username}", "password": "${password}"}
```

**Example connection string with secret name**

The following string has the secret name `${Test/RDS/MySql1}`.

```
mysql://jdbc:mysql://mysql1.host:3306/default?...&${Test/RDS/MySql1}&...
```

The connector uses the secret name to retrieve secrets and provide the user name and password, as in the following example.

```
mysql://jdbc:mysql://mysql1host:3306/default?...&user=sample2&password=sample2&...
```

Currently, the MySQL connector recognizes the user and password JDBC properties.

**Using a single connection handler**

You can use the following single connection metadata and record handlers to connect to a single MySQL instance.

<table>
<thead>
<tr>
<th>Handler type</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>MySqlCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>MySqlMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>MySqlRecordHandler</td>
</tr>
</tbody>
</table>

**Single connection handler parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when a catalog is not recognized.</td>
</tr>
</tbody>
</table>
The single connection handlers support one database instance and must provide a default connection string parameter. All other connection strings are ignored.

The following example property is for a single MySQL instance supported by a Lambda function.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>mysql://mysql1.host:3306/default?secret=Test/RDS/MySql1</td>
</tr>
</tbody>
</table>

**Spill parameters**

The Lambda SDK can spill data to Amazon S3. All database instances accessed by the same Lambda function spill to the same location.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spill_bucket</td>
<td>Required. Spill bucket name.</td>
</tr>
<tr>
<td>spill_prefix</td>
<td>Required. Spill bucket key prefix.</td>
</tr>
<tr>
<td>spill_put_request_headers</td>
<td>(Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, {&quot;x-amz-server-side-encryption&quot; : &quot;AES256&quot;}). For other possible headers, see PutObject in the Amazon Simple Storage Service API Reference.</td>
</tr>
</tbody>
</table>

**Data type support**

The following table shows the corresponding data types for JDBC and Arrow.

<table>
<thead>
<tr>
<th>JDBC</th>
<th>Arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Bit</td>
</tr>
<tr>
<td>Integer</td>
<td>Tiny</td>
</tr>
<tr>
<td>Short</td>
<td>Smallint</td>
</tr>
<tr>
<td>Integer</td>
<td>Int</td>
</tr>
<tr>
<td>Long</td>
<td>Bigint</td>
</tr>
<tr>
<td>float</td>
<td>Float4</td>
</tr>
<tr>
<td>Double</td>
<td>Float8</td>
</tr>
<tr>
<td>Date</td>
<td>DateDay</td>
</tr>
<tr>
<td>Timestamp</td>
<td>DateMilli</td>
</tr>
<tr>
<td>String</td>
<td>Varchar</td>
</tr>
<tr>
<td>Bytes</td>
<td>Varnbinary</td>
</tr>
<tr>
<td>BigDecimal</td>
<td>Decimal</td>
</tr>
</tbody>
</table>
Partitions and splits

Partitions are used to determine how to generate splits for the connector. Athena constructs a synthetic column of type `varchar` that represents the partitioning scheme for the table to help the connector generate splits. The connector does not modify the actual table definition.

Performance tuning

MySQL supports native partitions. The Athena Lambda connector can retrieve data from these partitions in parallel. If you want to query very large datasets with uniform partition distribution, native partitioning is highly recommended.

Predicate pushdown is performed within the Lambda function to decrease the data scanned by the query. However, selecting a subset of columns sometimes results in longer query execution runtime. LIMIT clauses reduce the amount of data scanned, but if a predicate is not provided, you should expect SELECT queries with a LIMIT clause to scan at least 16mb of data.

License information

By using this connector, you acknowledge the inclusion of third party components, a list of which can be found in the `pom.xml` file for this connector, and agree to the terms in the respective third party licenses provided in the `LICENSE.txt` file on GitHub.com.

See also

For the latest JDBC driver version information, see the `pom.xml` file for the MySQL connector on GitHub.com.

For additional information about this connector, visit the corresponding site on GitHub.com.

Amazon Athena Neptune connector

Amazon Neptune is a fast, reliable, fully managed graph database service that makes it easy to build and run applications that work with highly connected datasets. Neptune's purpose-built, high-performance graph database engine stores billions of relationships optimally and queries graphs with a latency of only milliseconds. For more information, see the Neptune User Guide.

The Amazon Athena Neptune Connector enables Athena to communicate with your Neptune graph database instance, making your Neptune graph data accessible by SQL queries.

Prerequisites

Using the Neptune connector requires the following three steps.

- Setting up a Neptune cluster
- Setting up an AWS Glue Data Catalog
- Deploying the connector to your AWS account. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64). For additional details specific to deploying the Neptune connector, see Deploy the Amazon Athena Neptune Connector on GitHub.com.
Limitations

Currently, the connector supports only the property graph model. RDF graphs are not currently supported.

Setting up a Neptune cluster

If you don’t have an existing Amazon Neptune cluster and property graph dataset in it that you would like to use, you must set one up.

Make sure you have an internet gateway and NAT gateway in the VPC that hosts your Neptune cluster. The private subnets that the Neptune connector Lambda function uses should have a route to the internet through this NAT Gateway. The Neptune connector Lambda function uses the NAT Gateway to communicate with AWS Glue.

For instructions on setting up a new Neptune cluster and loading it with a sample dataset, see Sample Neptune Cluster Setup on GitHub.com.

Setting up an AWS Glue Data Catalog

Unlike traditional relational data stores, Neptune graph DB nodes and edges do not use a set schema. Each entry can have different fields and data types. However, because the Neptune connector retrieves metadata from the AWS Glue Data Catalog, you must create an AWS Glue database that has tables with the required schema. After you create the AWS Glue database and tables, the connector can populate the list of tables available to query from Athena.

For information on setting up a AWS Glue Data Catalog to work with Neptune, see Set up AWS Glue Catalog on GitHub.com.

See also

For additional information about this connector, visit the corresponding site on GitHub.com.

Amazon Athena OpenSearch connector

The Amazon Athena OpenSearch connector enables Amazon Athena to communicate with your OpenSearch instances so that you can use SQL to query your OpenSearch data. This connector works with the Amazon OpenSearch Service or any OpenSearch-compatible endpoint that is configured with OpenSearch version 7.0 or higher.

Prerequisites

- Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).
- Set up a VPC and a security group before you use this connector. For more information, see Creating a VPC for a data source connector (p. 66).

Terms

The following terms relate to the OpenSearch connector.

- **Domain** – A name that this connector associates with the endpoint of your OpenSearch instance. The domain is also used as the database name. For OpenSearch instances defined within the Amazon
OpenSearch Service, the domain is auto-discoverable. For other instances, you must provide a mapping between the domain name and endpoint.

- **Index** – A database table defined in your OpenSearch instance.
- **Mapping** – If an index is a database table, then the mapping is its schema (that is, the definitions of its fields and attributes).

This connector supports both metadata retrieval from the OpenSearch instance and from the AWS Glue Data Catalog. If the connector finds a AWS Glue database and table that match your OpenSearch domain and index names, the connector attempts to use them for schema definition. We recommend that you create your AWS Glue table so that it is a superset of all fields in your OpenSearch index.

- **Document** – A record within a database table.

**Parameters**

Use the Lambda environment variables in this section to configure the OpenSearch connector.

- **spill_bucket** – Specifies the Amazon S3 bucket for data that exceeds Lambda function limits.
- **spill_prefix** – (Optional) Defaults to a subfolder in the specified spill_bucket called athena-federation-spill. We recommend that you configure an Amazon S3 storage lifecycle on this location to delete spills older than a predetermined number of days or hours.
- **spill_put_request_headers** – (Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, {"x-amz-server-side-encryption" : "AES256"}). For other possible headers, see PutObject in the Amazon Simple Storage Service API Reference.
- **kms_key_id** – (Optional) By default, any data that is spilled to Amazon S3 is encrypted using the AES-GCM authenticated encryption mode and a randomly generated key. To have your Lambda function use stronger encryption keys generated by KMS like a7e63k4b-81oc-40db-a2a1-4d0en2cd8331, you can specify a KMS key ID.
- **disable_spill_encryption** – (Optional) When set to True, disables spill encryption. Defaults to False so that data that is spilled to S3 is encrypted using AES-GCM – either using a randomly generated key or KMS to generate keys. Disabling spill encryption can improve performance, especially if your spill location uses server-side encryption.
- **disable_glue** – (Optional) If present and set to true, the connector does not attempt to retrieve supplemental metadata from AWS Glue.
- **query_timeout_cluster** – The timeout period, in seconds, for cluster health queries used in the generation of parallel scans.
- **query_timeout_search** – The timeout period, in seconds, for search queries used in the retrieval of documents from an index.
- **auto_discover_endpoint** – Boolean. The default is true. When you use the Amazon OpenSearch Service and set this parameter to true, the connector can auto-discover your domains and endpoints by calling the appropriate describe or list API operations on the OpenSearch Service. For any other type of OpenSearch instance (for example, self-hosted), you must specify the associated domain endpoints in the domain_mapping variable. If auto_discover_endpoint=true, the connector uses AWS credentials to authenticate to the OpenSearch Service. Otherwise, the connector retrieves user name and password credentials from AWS Secrets Manager through the domain_mapping variable.
- **domain_mapping** – Used only when auto_discover_endpoint is set to false and defines the mapping between domain names and their associated endpoints. The domain_mapping variable can accommodate multiple OpenSearch endpoints in the following format:

  domain1=endpoint1, domain2=endpoint2, domain3=endpoint3, ...

For the purpose of authenticating to an OpenSearch endpoint, the connector supports substitution strings injected using the format ${{SecretName}}: with user name and password retrieved from AWS.
Secrets Manager. The colon (:) at the end of the expression serves as a separator from the rest of the endpoint.

The following example uses the opensearch-creds secret.

```
movies=https://${opensearch-creds}:search-movies-ne...qu.us-east-1.es.amazonaws.com
```

At runtime, ${opensearch-creds} is rendered as the user name and password, as in the following example.

```
movies=https://myusername@mypassword:search-movies-ne...qu.us-east-1.es.amazonaws.com
```

In the domain_mapping parameter, each domain-endpoint pair can use a different secret. The secret itself must be specified in the format user_name@password. Although the password may contain embedded @ signs, the first @ serves as a separator from user_name.

It is also important to note that the comma (,) and equal sign (=) are used by this connector as separators for the domain-endpoint pairs. For this reason, you should not use them anywhere inside the stored secret.

**Setting up databases and tables in AWS Glue**

The connector obtains metadata information by using AWS Glue or OpenSearch. You can set up an AWS Glue table as a supplemental metadata definition source. To enable this feature, define a AWS Glue database and table that match the domain and index of the source that you are supplementing. The connector can also take advantage of metadata definitions stored in the OpenSearch instance by retrieving the mapping for the specified index.

**Defining metadata for arrays in OpenSearch**

OpenSearch does not have a dedicated array data type. Any field can contain zero or more values so long as they are of the same data type. If you want to use OpenSearch as your metadata definition source, you must define a _meta property for all indices used with Athena for the fields that to be considered a list or array. If you fail to complete this step, queries return only the first element in the list field. When you specify the _meta property, field names should be fully qualified for nested JSON structures (for example, address.street, where street is a nested field inside an address structure).

The following example defines actor and genre lists in the movies table.

```
PUT movies/_mapping
{
  "_meta": {
    "actor": "list",
    "genre": "list"
  }
}
```

**Data types**

The OpenSearch connector can extract metadata definitions from either AWS Glue or the OpenSearch instance. The connector uses the mapping in the following table to convert the definitions to Apache Arrow data types, including the points noted in the section that follows.

<table>
<thead>
<tr>
<th>OpenSearch</th>
<th>Apache Arrow</th>
<th>AWS Glue</th>
</tr>
</thead>
<tbody>
<tr>
<td>text, keyword, binary</td>
<td>VARCHAR</td>
<td>string</td>
</tr>
</tbody>
</table>
### Athena data source connectors

<table>
<thead>
<tr>
<th>OpenSearch</th>
<th>Apache Arrow</th>
<th>AWS Glue</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>BIGINT</td>
<td>bigint</td>
</tr>
<tr>
<td>scaled_float</td>
<td>BIGINT</td>
<td>SCALED_FLOAT(...)</td>
</tr>
<tr>
<td>integer</td>
<td>INT</td>
<td>int</td>
</tr>
<tr>
<td>short</td>
<td>SMALLINT</td>
<td>smallint</td>
</tr>
<tr>
<td>byte</td>
<td>TINYINT</td>
<td>tinyint</td>
</tr>
<tr>
<td>double</td>
<td>FLOAT8</td>
<td>double</td>
</tr>
<tr>
<td>float, half_float</td>
<td>FLOAT4</td>
<td>float</td>
</tr>
<tr>
<td>boolean</td>
<td>BIT</td>
<td>boolean</td>
</tr>
<tr>
<td>date, date_nanos</td>
<td>DATEMILLI</td>
<td>timestamp</td>
</tr>
<tr>
<td>JSON structure</td>
<td>STRUCT</td>
<td>STRUCT</td>
</tr>
<tr>
<td>_meta (for information, see the section Defining metadata for arrays in OpenSearch (p. 120).)</td>
<td>LIST</td>
<td>ARRAY</td>
</tr>
</tbody>
</table>

**Notes on data types**

- Currently, the connector supports only the OpenSearch and AWS Glue data-types listed in the preceding table.
- A *scaled_float* is a floating-point number scaled by a fixed double scaling factor and represented as a BIGINT in Apache Arrow. For example, 0.756 with a scaling factor of 100 is rounded to 76.
- To define a *scaled_float* in AWS Glue, you must select the *array* column type and declare the field using the format `SCALED_FLOAT(scaling_factor)`.

The following examples are valid:

```
SCALED_FLOAT(10.51)
SCALED_FLOAT(100)
SCALED_FLOAT(100.0)
```

The following examples are not valid:

```
SCALED_FLOAT(10.)
SCALED_FLOAT(.5)
```

- When converting from *date_nanos* to *DATEMILLI*, nanoseconds are rounded to the nearest millisecond. Valid values for *date* and *date_nanos* include, but are not limited to, the following formats:

```
"2020-05-18T10:15:30.123456789"
"2020-05-15T06:50:01.123Z"
"2020-05-15T06:49:30.123-05:00"
1589525370001 (epoch milliseconds)
```

- An OpenSearch binary is a string representation of a binary value encoded using Base64 and is converted to a VARCHAR.
Running SQL queries

The following are examples of DDL queries that you can use with this connector. In the examples, `function_name` corresponds to the name of your Lambda function, `domain` is the name of the domain that you want to query, and `index` is the name of your index.

- `SHOW DATABASES in 'lambda:function_name'`
- `SHOW TABLES in 'lambda:function_name'.domain`
- `DESCRIBE 'lambda:function_name'.domain.index`

Performance

The Athena OpenSearch connector supports shard-based parallel scans. The connector uses cluster health information retrieved from the OpenSearch instance to generate multiple requests for a document search query. The requests are split for each shard and run concurrently.

The connector also pushes down predicates as part of its document search queries. The following example query and predicate shows how the connector uses predicate push down.

Query

```
SELECT * FROM "lambda:elasticsearch".movies.movies
WHERE year >= 1955 AND year <= 1962 OR year = 1996
```

Predicate

```
(_exists_:year) AND year:([1955 TO 1962] OR 1996)
```

See also

For additional information about this connector, visit the corresponding site on GitHub.com.

Amazon Athena Oracle connector

The Amazon Athena connector for Oracle enables Amazon Athena to run SQL queries on data stored in Oracle running on-premises or on Amazon EC2 or Amazon RDS. You can also use the connector to query data on Oracle exadata.

Prerequisites

- Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).
- Set up a VPC and a security group before you use this connector. For more information, see Creating a VPC for a data source connector (p. 66).

Limitations

- Write DDL operations are not supported.
- In a multiplexer setup, the spill bucket and prefix are shared across all database instances.
• Any relevant Lambda limits. For more information, see Lambda quotas in the AWS Lambda Developer Guide.

Terms
The following terms relate to the Oracle connector.

• Database instance – Any instance of a database deployed on premises, on Amazon EC2, or on Amazon RDS.
• Handler – A Lambda handler that accesses your database instance. A handler can be for metadata or for data records.
• Metadata handler – A Lambda handler that retrieves metadata from your database instance.
• Record handler – A Lambda handler that retrieves data records from your database instance.
• Composite handler – A Lambda handler that retrieves both metadata and data records from your database instance.
• Property or parameter – A database property used by handlers to extract database information. You configure these properties as Lambda environment variables.
• Connection String – A string of text used to establish a connection to a database instance.
• Catalog – A non-AWS Glue catalog registered with Athena that is a required prefix for the connection_string property.
• Multiplexing handler – A Lambda handler that can accept and use multiple database connections.

Parameters
Use the Lambda environment variables in this section to configure the Oracle connector.

Connection string
Use a JDBC connection string in the following format to connect to a database instance.

oracle://${jdbc_connection_string}

Using a multiplexing handler
You can use a multiplexer to connect to multiple database instances with a single Lambda function. Requests are routed by catalog name. Use the following classes in Lambda.

<table>
<thead>
<tr>
<th>Handler</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>OracleMuxCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>OracleMuxMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>OracleMuxRecordHandler</td>
</tr>
</tbody>
</table>

Multiplexing handler parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$catalog_connection_string</td>
<td>Required. A database instance connection string. Prefix the string with the name of the catalog used in Athena. For example, if the catalog registered with Athena is myoraclecatalog, then the environment variable name is myoraclecatalog_connection_string.</td>
</tr>
</tbody>
</table>
### Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when the catalog is lambda: <code>${AWS_LAMBDA_FUNCTION_NAME}</code>.</td>
</tr>
</tbody>
</table>

The following example properties are for a Oracle MUX Lambda function that supports two database instances: oracle1 (the default), and oracle2.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>oracle://jdbc:oracle:thin:${Test/RDS/Oracle1}@//oracle1.hostname:port/servicename</td>
</tr>
<tr>
<td>oracle_catalog1_connection_string</td>
<td>oracle://jdbc:oracle:thin:${Test/RDS/Oracle1}@//oracle1.hostname:port/servicename</td>
</tr>
</tbody>
</table>

### Providing credentials

To provide a user name and password for your database in your JDBC connection string, you can use connection string properties or AWS Secrets Manager.

- **Connection String** – A user name and password can be specified as properties in the JDBC connection string.

- **AWS Secrets Manager** – To use the Athena Federated Query feature with AWS Secrets Manager, the VPC connected to your Lambda function should have internet access or a VPC endpoint to connect to Secrets Manager.

You can put the name of a secret in AWS Secrets Manager in your JDBC connection string. The connector replaces the secret name with the username and password values from Secrets Manager.

For Amazon RDS database instances, this support is tightly integrated. If you use Amazon RDS, we highly recommend using AWS Secrets Manager and credential rotation. If your database does not use Amazon RDS, store the credentials as JSON in the following format:

```json
{"username": "${username}", "password": "${password}"}
```

### Example connection string with secret name

The following string has the secret name ${Test/RDS/Oracle}.

```
oracle://jdbc:oracle:thin:${Test/RDS/Oracle}@//hostname:port/servicename
```

The connector uses the secret name to retrieve secrets and provide the user name and password, as in the following example.

```
```
Currently, the Oracle connector recognizes the UID and PWD JDBC properties.

**Using a single connection handler**

You can use the following single connection metadata and record handlers to connect to a single Oracle instance.

<table>
<thead>
<tr>
<th>Handler type</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>OracleCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>OracleMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>OracleRecordHandler</td>
</tr>
</tbody>
</table>

**Single connection handler parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when a catalog is not recognized.</td>
</tr>
</tbody>
</table>

The single connection handlers support one database instance and must provide a default connection string parameter. All other connection strings are ignored.

The connector supports SSL based connections to Amazon RDS instances. Support is limited to the Transport Layer Security (TLS) protocol and to authentication of the server by the client. Mutual authentication it is not supported in Amazon RDS. The second row in the table below shows the syntax for using SSL.

The following example property is for a single Oracle instance supported by a Lambda function.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>oracle://jdbc:oracle:thin:${Test/RDS/Oracle}@//$hostname:port/$servicename</td>
</tr>
</tbody>
</table>

**Spill parameters**

The Lambda SDK can spill data to Amazon S3. All database instances accessed by the same Lambda function spill to the same location.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spill_bucket</td>
<td>Required. Spill bucket name.</td>
</tr>
<tr>
<td>spill_prefix</td>
<td>Required. Spill bucket key prefix.</td>
</tr>
</tbody>
</table>
### Data type support

The following table shows the corresponding data types for JDBC, Oracle, and Arrow.

<table>
<thead>
<tr>
<th>JDBC</th>
<th>Oracle</th>
<th>Arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>boolean</td>
<td>Bit</td>
</tr>
<tr>
<td>Integer</td>
<td>N/A</td>
<td>Tiny</td>
</tr>
<tr>
<td>Short</td>
<td>smallint</td>
<td>Smallint</td>
</tr>
<tr>
<td>Integer</td>
<td>integer</td>
<td>Int</td>
</tr>
<tr>
<td>Long</td>
<td>bigint</td>
<td>Bigint</td>
</tr>
<tr>
<td>float</td>
<td>float4</td>
<td>Float4</td>
</tr>
<tr>
<td>Double</td>
<td>float8</td>
<td>Float8</td>
</tr>
<tr>
<td>Date</td>
<td>date</td>
<td>DateDay</td>
</tr>
<tr>
<td>Timestamp</td>
<td>timestamp</td>
<td>DateMilli</td>
</tr>
<tr>
<td>String</td>
<td>text</td>
<td>Varchar</td>
</tr>
<tr>
<td>Bytes</td>
<td>bytes</td>
<td>Varbinary</td>
</tr>
<tr>
<td>BigDecimal</td>
<td>numeric(p,s)</td>
<td>Decimal</td>
</tr>
<tr>
<td>ARRAY</td>
<td>N/A (see note)</td>
<td>List</td>
</tr>
</tbody>
</table>

### Partitions and splits

Partitions are used to determine how to generate splits for the connector. Athena constructs a synthetic column of type `varchar` that represents the partitioning scheme for the table to help the connector generate splits. The connector does not modify the actual table definition.

### Performance tuning

Oracle supports native partitions. The Athena Lambda connector can retrieve data from these partitions in parallel. If you want to query very large datasets with uniform partition distribution, native partitioning is highly recommended.

### License information

By using this connector, you acknowledge the inclusion of third party components, a list of which can be found in the `pom.xml` file for this connector, and agree to the terms in the respective third party licenses provided in the `LICENSE.txt` file on GitHub.com.
See also

For the latest JDBC driver version information, see the pom.xml file for the Oracle connector on GitHub.com.

For additional information about this connector, visit the corresponding site on GitHub.com.

Amazon Athena PostgreSQL connector

The Amazon Athena PostgreSQL connector enables Athena to access your PostgreSQL databases.

Prerequisites

- Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).

Limitations

- Write DDL operations are not supported.
- In a multiplexer setup, the spill bucket and prefix are shared across all database instances.
- Any relevant Lambda limits. For more information, see Lambda quotas in the AWS Lambda Developer Guide.

Terms

The following terms relate to the PostgreSQL connector.

- **Database instance** – Any instance of a database deployed on premises, on Amazon EC2, or on Amazon RDS.
- **Handler** – A Lambda handler that accesses your database instance. A handler can be for metadata or for data records.
- **Metadata handler** – A Lambda handler that retrieves metadata from your database instance.
- **Record handler** – A Lambda handler that retrieves data records from your database instance.
- **Composite handler** – A Lambda handler that retrieves both metadata and data records from your database instance.
- **Property or parameter** – A database property used by handlers to extract database information. You configure these properties as Lambda environment variables.
- **Connection String** – A string of text used to establish a connection to a database instance.
- **Catalog** – A non-AWS Glue catalog registered with Athena that is a required prefix for the connection_string property.
- **Multiplexing handler** – A Lambda handler that can accept and use multiple database connections.

Parameters

Use the Lambda environment variables in this section to configure the PostgreSQL connector.

**Connection string**

Use a JDBC connection string in the following format to connect to a database instance.
Using a multiplexing handler

You can use a multiplexer to connect to multiple database instances with a single Lambda function. Requests are routed by catalog name. Use the following classes in Lambda.

<table>
<thead>
<tr>
<th>Handler</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>PostGreSqlMuxCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>PostGreSqlMuxMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>PostGreSqlMuxRecordHandler</td>
</tr>
</tbody>
</table>

Multiplexing handler parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$catalog_connection_string</td>
<td>Required. A database instance connection string. Prefix the string with the name of the catalog used in Athena. For example, if the catalog registered with Athena is mypostgresqlcatalog, then the environment variable name is mypostgresqlcatalog_connection_string.</td>
</tr>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when the catalog is lambda:$AWS_LAMBDA_FUNCTION_NAME.</td>
</tr>
</tbody>
</table>

The following example properties are for a PostGreSql MUX Lambda function that supports two database instances: postgres1 (the default), and postgres2.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>postgres://jdbc:postgresql://postgres1.host:5432/default?${Test/RDS/Postgres1}</td>
</tr>
<tr>
<td>postgres_catalog1_connection_string</td>
<td>postgres://jdbc:postgresql://postgres1.host:5432/default?${Test/RDS/Postgres1}</td>
</tr>
</tbody>
</table>

Providing credentials

To provide a user name and password for your database in your JDBC connection string, you can use connection string properties or AWS Secrets Manager.

- **Connection String** – A user name and password can be specified as properties in the JDBC connection string.
- **AWS Secrets Manager** – To use the Athena Federated Query feature with AWS Secrets Manager, the VPC connected to your Lambda function should have internet access or a VPC endpoint to connect to Secrets Manager.
You can put the name of a secret in AWS Secrets Manager in your JDBC connection string. The connector replaces the secret name with the username and password values from Secrets Manager.

For Amazon RDS database instances, this support is tightly integrated. If you use Amazon RDS, we highly recommend using AWS Secrets Manager and credential rotation. If your database does not use Amazon RDS, store the credentials as JSON in the following format:

```
{"username": "$\{username\}", "password": "$\{password\}"
```

**Example connection string with secret name**

The following string has the secret name ${Test/RDS/PostGres1}.

```
postgres://jdbc:postgres://postgres1.host:3306/default?...&${Test/RDS/PostGres1}&...
```

The connector uses the secret name to retrieve secrets and provide the user name and password, as in the following example.

```
postgres://jdbc:postgres://postgres1.host:3306/
default?...&user=sample2&password=sample2&...
```

Currently, the PostgreSQL connector recognizes the user and password JDBC properties.

**Using a single connection handler**

You can use the following single connection metadata and record handlers to connect to a single PostgreSQL instance.

<table>
<thead>
<tr>
<th>Handler type</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>PostGreSqlCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>PostGreSqlMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>PostGreSqlRecordHandler</td>
</tr>
</tbody>
</table>

**Single connection handler parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when a catalog is not recognized.</td>
</tr>
</tbody>
</table>

The single connection handlers support one database instance and must provide a default connection string parameter. All other connection strings are ignored.

The following example property is for a single PostgreSQL instance supported by a Lambda function.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>postgres://jdbc:postgresql://postgres1.host:3306/default?secret=${Test/RDS/PostgreSQL1}</td>
</tr>
</tbody>
</table>
Spill parameters

The Lambda SDK can spill data to Amazon S3. All database instances accessed by the same Lambda function spill to the same location.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spill_bucket</td>
<td>Required. Spill bucket name.</td>
</tr>
<tr>
<td>spill_prefix</td>
<td>Required. Spill bucket key prefix.</td>
</tr>
<tr>
<td>spill_put_request_headers</td>
<td>(Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, {&quot;x-amz-server-side-encryption&quot;: &quot;AES256&quot;}). For other possible headers, see PutObject in the Amazon Simple Storage Service API Reference.</td>
</tr>
</tbody>
</table>

Data type support

The following table shows the corresponding data types for JDBC, PostGreSQL, and Arrow.

<table>
<thead>
<tr>
<th>JDBC</th>
<th>PostGreSQL</th>
<th>Arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Boolean</td>
<td>Bit</td>
</tr>
<tr>
<td>Integer</td>
<td>N/A</td>
<td>Tiny</td>
</tr>
<tr>
<td>Short</td>
<td>smallint</td>
<td>Smallint</td>
</tr>
<tr>
<td>Integer</td>
<td>integer</td>
<td>Int</td>
</tr>
<tr>
<td>Long</td>
<td>bigint</td>
<td>Bigint</td>
</tr>
<tr>
<td>float</td>
<td>float4</td>
<td>Float4</td>
</tr>
<tr>
<td>Double</td>
<td>float8</td>
<td>Float8</td>
</tr>
<tr>
<td>Date</td>
<td>date</td>
<td>DateDay</td>
</tr>
<tr>
<td>Timestamp</td>
<td>timestamp</td>
<td>DateMilli</td>
</tr>
<tr>
<td>String</td>
<td>text</td>
<td>Varchar</td>
</tr>
<tr>
<td>Bytes</td>
<td>bytes</td>
<td>Varbinary</td>
</tr>
<tr>
<td>BigDecimal</td>
<td>numeric(p,s)</td>
<td>Decimal</td>
</tr>
<tr>
<td>ARRAY</td>
<td>N/A (see note)</td>
<td>List</td>
</tr>
</tbody>
</table>

**Note**

The ARRAY type is supported for the PostgreSQL connector with the following constraints: Multidimensional arrays (<data_type>[][] or nested arrays) are not supported. Columns with unsupported ARRAY data-types are converted to an array of string elements (array<varchar>).
Partitions and splits

Partitions are used to determine how to generate splits for the connector. Athena constructs a synthetic column of type `varchar` that represents the partitioning scheme for the table to help the connector generate splits. The connector does not modify the actual table definition.

Performance tuning

PostgreSQL supports native partitions. The Athena Lambda connector can retrieve data from these partitions in parallel. If you want to query very large datasets with uniform partition distribution, native partitioning is highly recommended.

See also

For the latest JDBC driver version information, see the `pom.xml` file for the PostgreSQL connector on GitHub.com.

For additional information about this connector, visit the corresponding site on GitHub.com.

Amazon Athena Redis connector

The Amazon Athena Redis connector enables Amazon Athena to communicate with your Redis instances so that you can query your Redis data with SQL. You can use the AWS Glue Data Catalog to map your Redis key-value pairs into virtual tables.

Unlike traditional relational data stores, Redis does not have the concept of a table or a column. Instead, Redis offers key-value access patterns where the key is essentially a string and the value is a string, z-set, or hmap.

You can use the AWS Glue Data Catalog to create schema and configure virtual tables. Special table properties tell the Athena Redis connector how to map your Redis keys and values into a table. For more information, see Setting up databases and tables in AWS Glue (p. 132) later in this document.

Prerequisites

- Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).
- Set up a VPC and a security group before you use this connector. For more information, see Creating a VPC for a data source connector (p. 66).

Parameters

Use the Lambda environment variables in this section to configure the Redis connector.

- `spill_bucket` – Specifies the Amazon S3 bucket for data that exceeds Lambda function limits.
- `spill_prefix` – (Optional) Defaults to a subfolder in the specified `spill_bucket` called athena-federation-spill. We recommend that you configure an Amazon S3 storage lifecycle on this location to delete spills older than a predetermined number of days or hours.
- `spill_put_request_headers` – (Optional) A JSON encoded map of request headers and values for the Amazon S3 `putObject` request that is used for spilling (for example, `{"x-amz-server-side-encryption" : "AES256"}`). For other possible headers, see `PutObject` in the Amazon Simple Storage Service API Reference.
- `kms_key_id` – (Optional) By default, any data that is spilled to Amazon S3 is encrypted using the AES-GCM authenticated encryption mode and a randomly generated key. To have your Lambda function...
use stronger encryption keys generated by KMS like a7e63k4b-8loc-40db-a2a1-4d0en2cd8331, you can specify a KMS key ID.

- **disable_spill_encryption** – (Optional) When set to True, disables spill encryption. Defaults to False so that data that is spilled to S3 is encrypted using AES-GCM – either using a randomly generated key or KMS to generate keys. Disabling spill encryption can improve performance, especially if your spill location uses server-side encryption.

- **glue_catalog** – (Optional) Use this option to specify a cross-account AWS Glue catalog (p. 24). By default, the connector attempts to get metadata from its own AWS Glue account.

### Setting up databases and tables in AWS Glue

To enable an AWS Glue table for use with Redis, you can set the following table properties on the table: `redis-endpoint`, `redis-value-type`, and either `redis-keys-zset` or `redis-key-prefix`.

In addition, any AWS Glue database that contains Redis tables must have a `redis-db-flag` in the URI property of the database. To set the `redis-db-flag` URI property, use the AWS Glue console to edit the database.

The following list describes the table properties.

- **redis-endpoint** – (Required) The `hostname:port:password` of the Redis server that contains data for this table (for example, athena-federation-demo.cache.amazonaws.com:6379). Alternatively, you can store the endpoint, or part of the endpoint, in AWS Secrets Manager by using `${Secret_Name}` as the table property value.

  **Note**
  To use the Athena Federated Query feature with AWS Secrets Manager, the VPC connected to your Lambda function should have internet access or a VPC endpoint to connect to Secrets Manager.

- **redis-keys-zset** – (Required if `redis-key-prefix` is not used) A comma-separated list of keys whose value is a zset (for example, active-orders,pending-orders). Each of the values in the zset is treated as a key that is part of the table. Either the `redis-keys-zset` property or the `redis-key-prefix` property must be set.

- **redis-key-prefix** – (Required if `redis-keys-zset` is not used) A comma separated list of key prefixes to scan for values in the table (for example, accounts-*), acct-). Either the `redis-key-prefix` property or the `redis-keys-zset` property must be set.

- **redis-value-type** – (Required) Defines how the values for the keys defined by either `redis-key-prefix` or `redis-keys-zset` are mapped to your table. A literal maps to a single column. A zset also maps to a single column, but each key can store many rows. A hash enables each key to be a row with multiple columns (for example, a hash, literal, or zset.)

- **redis-ssl-flag** – (Optional) When True, creates a Redis connection that uses SSL/TLS. The default is False.

- **redis-cluster-flag** – (Optional) When True, enables support for clustered Redis instances. The default is False.

- **redis-db-number** – (Optional) Applies only to standalone, non-clustered instances.) Set this number (for example 1, 2, or 3) to read from a non-default Redis database. The default is Redis logical database 0. This number does not refer to a database in Athena or AWS Glue, but to a Redis logical database. For more information, see `SELECT index` in the Redis documentation.

### Data types

All Redis values are retrieved as the string data type. Then they are converted to one of the following Apache Arrow data types based on how your tables are defined in the AWS Glue Data Catalog.
<table>
<thead>
<tr>
<th>AWS Glue data type</th>
<th>Apache Arrow data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>INT</td>
</tr>
<tr>
<td>string</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>bigint</td>
<td>BIGINT</td>
</tr>
<tr>
<td>double</td>
<td>FLOAT8</td>
</tr>
<tr>
<td>float</td>
<td>FLOAT4</td>
</tr>
<tr>
<td>smallint</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>tinyint</td>
<td>TINYINT</td>
</tr>
<tr>
<td>boolean</td>
<td>BIT</td>
</tr>
<tr>
<td>binary</td>
<td>VARBINARY</td>
</tr>
</tbody>
</table>

**Required Permissions**

Review the Policies section of the `athena-redis.yaml` file for full details on the IAM policies that this connector requires. The following is a brief summary.

- **Amazon S3 write access** – To successfully handle large queries, the connector requires write access to a location in Amazon S3.
- **Athena GetQueryExecution** – The connector uses this permission to fast-fail when the upstream Athena query has terminated.
- **AWS Glue Data Catalog** – The Redis connector requires read only access to the AWS Glue Data Catalog to obtain schema information.
- **CloudWatch Logs** – The connector requires access to CloudWatch Logs for storing logs.
- **AWS Secrets Manager read access** – If you choose to store Redis endpoint details in Secrets Manager, you must grant the connector access to those secrets.
- **VPC access** – The connector requires the ability to attach and detach interfaces to your VPC so that it can connect to it and communicate with your Redis instances.

**Performance**

The Athena Redis connector attempts to parallelize queries against your Redis instance according to the type of table that you have defined (for example, zset keys or prefix keys). Predicate pushdown is performed within the Lambda function.

**License information**

The Amazon Athena Redis connector project is licensed under the Apache-2.0 License.

**See also**

For additional information about this connector, visit the corresponding site on GitHub.com.

**Amazon Athena Redshift connector**

The Amazon Athena Redshift connector enables Amazon Athena to access your Amazon Redshift databases.
Prerequisites

- Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).

Limitations

- Write DDL operations are not supported.
- In a multiplexer setup, the spill bucket and prefix are shared across all database instances.
- Any relevant Lambda limits. For more information, see Lambda quotas in the AWS Lambda Developer Guide.
- Because Redshift does not support external partitions, all data specified by a query is retrieved every time.

Terms

The following terms relate to the Redshift connector.

- **Database instance** – Any instance of a database deployed on premises, on Amazon EC2, or on Amazon RDS.
- **Handler** – A Lambda handler that accesses your database instance. A handler can be for metadata or for data records.
- **Metadata handler** – A Lambda handler that retrieves metadata from your database instance.
- **Record handler** – A Lambda handler that retrieves data records from your database instance.
- **Composite handler** – A Lambda handler that retrieves both metadata and data records from your database instance.
- **Property or parameter** – A database property used by handlers to extract database information. You configure these properties as Lambda environment variables.
- **Connection String** – A string of text used to establish a connection to a database instance.
- **Catalog** – A non-AWS Glue catalog registered with Athena that is a required prefix for the connection_string property.
- **Multiplexing handler** – A Lambda handler that can accept and use multiple database connections.

Parameters

Use the Lambda environment variables in this section to configure the Redshift connector.

Connection string

Use a JDBC connection string in the following format to connect to a database instance.

```
redshift://${jdbc_connection_string}
```

Using a multiplexing handler

You can use a multiplexer to connect to multiple database instances with a single Lambda function. Requests are routed by catalog name. Use the following classes in Lambda.
### Handler Class

<table>
<thead>
<tr>
<th>Handler</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>RedshiftMuxCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>RedshiftMuxMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>RedshiftMuxRecordHandler</td>
</tr>
</tbody>
</table>

### Multiplexing handler parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$catalog_connection_string</td>
<td>Required. A database instance connection string. Prefix the string with the name of the catalog used in Athena. For example, if the catalog registered with Athena is myredshiftcatalog, then the environment variable name is myredshiftcatalog_connection_string.</td>
</tr>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when the catalog is lambda:${AWS_LAMBDA_FUNCTION_NAME}.</td>
</tr>
</tbody>
</table>

The following example properties are for a Redshift MUX Lambda function that supports two database instances: `redshift1` (the default), and `redshift2`.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td><code>redshift://jdbc:redshift://redshift1.host:5439/dev?user=sample2&amp;password=sample2</code></td>
</tr>
<tr>
<td>redshift_catalog1_connection_string</td>
<td><code>redshift://jdbc:redshift://redshift1.host:3306/default?${Test/RDS/Redshift1}</code></td>
</tr>
<tr>
<td>redshift_catalog2_connection_string</td>
<td><code>redshift://jdbc:redshift://redshift2.host:3333/default?user=sample2&amp;password=sample2</code></td>
</tr>
</tbody>
</table>

### Providing credentials

To provide a user name and password for your database in your JDBC connection string, you can use connection string properties or AWS Secrets Manager.

- **Connection String** – A user name and password can be specified as properties in the JDBC connection string.

- **AWS Secrets Manager** – To use the Athena Federated Query feature with AWS Secrets Manager, the VPC connected to your Lambda function should have internet access or a VPC endpoint to connect to Secrets Manager.

You can put the name of a secret in AWS Secrets Manager in your JDBC connection string. The connector replaces the secret name with the username and password values from Secrets Manager.
For Amazon RDS database instances, this support is tightly integrated. If you use Amazon RDS, we highly recommend using AWS Secrets Manager and credential rotation. If your database does not use Amazon RDS, store the credentials as JSON in the following format:

```json
{"username": "$\{username\}$", "password": "$\{password\}$"}
```

**Example connection string with secret name**

The following string has the secret name ${Test/RDS/ Redshift1}.

```bash
redshift://jdbc:redshift://redshift1.host:3306/default?...&${Test/RDS/Redshift1}&... 
```

The connector uses the secret name to retrieve secrets and provide the user name and password, as in the following example.

```bash
redshift://jdbc:redshift://redshift1.host:3306/default?...&user=sample2&password=sample2&... 
```

Currently, the Redshift connector recognizes the user and password JDBC properties.

**Using a single connection handler**

You can use the following single connection metadata and record handlers to connect to a single Redshift instance.

<table>
<thead>
<tr>
<th>Handler type</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>RedshiftCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>RedshiftMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>RedshiftRecordHandler</td>
</tr>
</tbody>
</table>

**Single connection handler parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when a catalog is not recognized.</td>
</tr>
</tbody>
</table>

The single connection handlers support one database instance and must provide a default connection string parameter. All other connection strings are ignored.

The following example property is for a single Redshift instance supported by a Lambda function.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
</table>
Spill parameters

The Lambda SDK can spill data to Amazon S3. All database instances accessed by the same Lambda function spill to the same location.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spill_bucket</td>
<td>Required. Spill bucket name.</td>
</tr>
<tr>
<td>spill_prefix</td>
<td>Required. Spill bucket key prefix.</td>
</tr>
<tr>
<td>spill_put_request_headers</td>
<td>(Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, {&quot;x-amz-server-side-encryption&quot; : &quot;AES256&quot;}). For other possible headers, see PutObject in the Amazon Simple Storage Service API Reference.</td>
</tr>
</tbody>
</table>

Data type support

The following table shows the corresponding data types for JDBC and Apache Arrow.

<table>
<thead>
<tr>
<th>JDBC</th>
<th>Arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Bit</td>
</tr>
<tr>
<td>Integer</td>
<td>Tiny</td>
</tr>
<tr>
<td>Short</td>
<td>Smallint</td>
</tr>
<tr>
<td>Integer</td>
<td>Int</td>
</tr>
<tr>
<td>Long</td>
<td>Bigint</td>
</tr>
<tr>
<td>Float</td>
<td>Float4</td>
</tr>
<tr>
<td>Double</td>
<td>Float8</td>
</tr>
<tr>
<td>Date</td>
<td>DateDay</td>
</tr>
<tr>
<td>Timestamp</td>
<td>DateMilli</td>
</tr>
<tr>
<td>String</td>
<td>Varchar</td>
</tr>
<tr>
<td>Bytes</td>
<td>Varbinary</td>
</tr>
<tr>
<td>BigDecimal</td>
<td>Decimal</td>
</tr>
<tr>
<td>ARRAY</td>
<td>List</td>
</tr>
</tbody>
</table>

Partitions and splits

Redshift does not support external partitions. For information about performance related issues, see Performance tuning (p. 137).

Performance tuning

Predicate pushdown is performed within the Lambda function to decrease the data scanned by the query. However, selecting a subset of columns sometimes results in longer query execution runtime.
LIMIT clauses reduce the amount of data scanned, but if a predicate is not provided, you should expect SELECT queries with a LIMIT clause to scan at least 16mb of data.

Amazon Redshift is particularly susceptible to query execution slowdown when you run multiple queries concurrently. For example, in one test, when concurrency was changed from 1 query to 8 queries, latency went from 17 seconds to 800 seconds for a simple SELECT * LIMIT 1 query.

See also

For the latest JDBC driver version information, see the pom.xml file for the Redshift connector on GitHub.com.

For additional information about this connector, visit the corresponding site on GitHub.com.

Amazon Athena SAP HANA connector

Prerequisites

- Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).
- Set up a VPC and a security group before you use this connector. For more information, see Creating a VPC for a data source connector (p. 66).

Limitations

- Write DDL operations are not supported.
- In a multiplexer setup, the spill bucket and prefix are shared across all database instances.
- Any relevant Lambda limits. For more information, see Lambda quotas in the AWS Lambda Developer Guide.

In SAP HANA, object names are converted to uppercase when they are stored in the SAP HANA database. However, because names in quotation marks are case sensitive, it is possible for two tables to have the same name in lower and upper case (for example, EMPLOYEE and employee).

In Athena Federated Query, schema table names are provided to the Lambda function in lower case. To work around this issue, you can provide @schemaCase query hints to retrieve the data from the tables that have case sensitive names. Following are two sample queries with query hints.

```
SELECT *
FROM "lambda:saphanaconnector".SYSTEM."MY_TABLE@schemaCase=upper&tableCase=upper"
```

```
SELECT *
FROM "lambda:saphanaconnector".SYSTEM."MY_TABLE@schemaCase=upper&tableCase=lower"
```

Terms

The following terms relate to the SAP HANA connector.

- **Database instance** – Any instance of a database deployed on premises, on Amazon EC2, or on Amazon RDS.
- **Handler** – A Lambda handler that accesses your database instance. A handler can be for metadata or for data records.
- **Metadata handler** – A Lambda handler that retrieves metadata from your database instance.
• **Record handler** – A Lambda handler that retrieves data records from your database instance.
• **Composite handler** – A Lambda handler that retrieves both metadata and data records from your database instance.
• **Property or parameter** – A database property used by handlers to extract database information. You configure these properties as Lambda environment variables.
• **Connection String** – A string of text used to establish a connection to a database instance.
• **Catalog** – A non-AWS Glue catalog registered with Athena that is a required prefix for the connection_string property.
• **Multiplexing handler** – A Lambda handler that can accept and use multiple database connections.

**Parameters**

Use the Lambda environment variables in this section to configure the SAP HANA connector.

**Connection string**

Use a JDBC connection string in the following format to connect to a database instance.

```
saphana://${jdbc_connection_string}
```

**Using a multiplexing handler**

You can use a multiplexer to connect to multiple database instances with a single Lambda function. Requests are routed by catalog name. Use the following classes in Lambda.

<table>
<thead>
<tr>
<th>Handler</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>SaphanaMuxCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>SaphanaMuxMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>SaphanaMuxRecordHandler</td>
</tr>
</tbody>
</table>

**Multiplexing handler parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$catalog_connection_string</td>
<td>Required. A database instance connection string. Prefix the string with the name of the catalog used in Athena. For example, if the catalog registered with Athena is mysaphanacatalog, then the environment variable name is mysaphanacatalog_connection_string.</td>
</tr>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when the catalog is lambda:${AWS_LAMBDA_FUNCTION_NAME}.</td>
</tr>
</tbody>
</table>

The following example properties are for a Saphana MUX Lambda function that supports two database instances: saphanain (the default), and saphana2.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>saphana://jdbc:sap://saphana1.host:port/?${Test/RDS/Saphana1}</td>
</tr>
</tbody>
</table>
Providing credentials

To provide a user name and password for your database in your JDBC connection string, you can use connection string properties or AWS Secrets Manager.

- **Connection String** – A user name and password can be specified as properties in the JDBC connection string.
- **AWS Secrets Manager** – To use the Athena Federated Query feature with AWS Secrets Manager, the VPC connected to your Lambda function should have internet access or a VPC endpoint to connect to Secrets Manager.

You can put the name of a secret in AWS Secrets Manager in your JDBC connection string. The connector replaces the secret name with the username and password values from Secrets Manager.

For Amazon RDS database instances, this support is tightly integrated. If you use Amazon RDS, we highly recommend using AWS Secrets Manager and credential rotation. If your database does not use Amazon RDS, store the credentials as JSON in the following format:

```json
{"username": "${username}", "password": "${password}"}
```

Example connection string with secret name

The following string has the secret name `${Test/RDS/Saphana1}`.

```
saphana://jdbc:sap://saphana1.host:port/?${Test/RDS/Saphana1}&...
```

The connector uses the secret name to retrieve secrets and provide the user name and password, as in the following example.

```
saphana://jdbc:sap://saphana1.host:port/?user=sample2&password=sample2&...
```

Currently, the SAP HANA connector recognizes the user and password JDBC properties.

Using a single connection handler

You can use the following single connection metadata and record handlers to connect to a single SAP HANA instance.

<table>
<thead>
<tr>
<th>Handler type</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>SaphanaCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>SaphanaMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>SaphanaRecordHandler</td>
</tr>
</tbody>
</table>
Single connection handler parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when a catalog is not recognized.</td>
</tr>
</tbody>
</table>

The single connection handlers support one database instance and must provide a default connection string parameter. All other connection strings are ignored.

The following example property is for a single SAP HANA instance supported by a Lambda function.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>saphana://jdbc:sap://saphana1.host:port/?secret=Test/RDS/Saphana1</td>
</tr>
</tbody>
</table>

Spill parameters

The Lambda SDK can spill data to Amazon S3. All database instances accessed by the same Lambda function spill to the same location.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spill_bucket</td>
<td>Required. Spill bucket name.</td>
</tr>
<tr>
<td>spill_prefix</td>
<td>Required. Spill bucket key prefix.</td>
</tr>
<tr>
<td>spill_put_request_headers</td>
<td>(Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, {&quot;x-amz-server-side-encryption&quot;: &quot;AES256&quot;}). For other possible headers, see PutObject in the Amazon Simple Storage Service API Reference.</td>
</tr>
</tbody>
</table>

Data type support

The following table shows the corresponding data types for JDBC and Apache Arrow.

<table>
<thead>
<tr>
<th>JDBC</th>
<th>Arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Bit</td>
</tr>
<tr>
<td>Integer</td>
<td>Tiny</td>
</tr>
<tr>
<td>Short</td>
<td>Smallint</td>
</tr>
<tr>
<td>Integer</td>
<td>Int</td>
</tr>
<tr>
<td>Long</td>
<td>Bigint</td>
</tr>
<tr>
<td>float</td>
<td>Float4</td>
</tr>
<tr>
<td>Double</td>
<td>Float8</td>
</tr>
</tbody>
</table>
Data type conversions

In addition to the JDBC to Arrow conversions, the connector performs certain other conversions to make the SAP HANA source and Athena data types compatible. These conversions help ensure that queries get executed successfully. The following table shows these conversions.

<table>
<thead>
<tr>
<th>Source data type (SAP HANA)</th>
<th>Converted data type (Athena)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECIMAL</td>
<td>BIGINT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INT</td>
</tr>
<tr>
<td>DATE</td>
<td>DATEDAY</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>DATEMILLI</td>
</tr>
</tbody>
</table>

All other unsupported data types are converted to VARCHAR.

Partitions and splits

A partition is represented by a single partition column of type Integer. The column contains partition names of the partitions defined on an SAP HANA table. For a table that does not have partition names, * is returned, which is equivalent to a single partition. A partition is equivalent to a split.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PART_ID</td>
<td>Integer</td>
<td>Named partition in SAP HANA.</td>
</tr>
</tbody>
</table>

Performance tuning

SAP HANA supports native partitions. The Athena Lambda connector can retrieve data from these partitions in parallel. If you want to query very large datasets with uniform partition distribution, native partitioning is highly recommended.

License information

By using this connector, you acknowledge the inclusion of third party components, a list of which can be found in the pom.xml file for this connector, and agree to the terms in the respective third party licenses provided in the LICENSE.txt file on GitHub.com.
See also

For the latest JDBC driver version information, see the pom.xml file for the SAP HANA connector on GitHub.com.

For additional information about this connector, visit the corresponding site on GitHub.com.

Amazon Athena Snowflake connector

The Amazon Athena connector for Snowflake enables Amazon Athena to run SQL queries on data stored in your Snowflake SQL database or RDS instances using JDBC.

Prerequisites

• Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).

• Set up a VPC and a security group before you use this connector. For more information, see Creating a VPC for a data source connector (p. 66).

Limitations

• Write DDL operations are not supported.

• In a multiplexer setup, the spill bucket and prefix are shared across all database instances.

• Any relevant Lambda limits. For more information, see Lambda quotas in the AWS Lambda Developer Guide.

• Currently, Snowflake views are not supported.

• In Snowflake, because object names are case sensitive, two tables can have the same name in lower and upper case (for example, EMPLOYEE and employee). In Athena Federated Query, schema table names are provided to the Lambda function in lower case. To work around this issue, you can provide @schemaCase query hints to retrieve the data from the tables that have case sensitive names. Following are two sample queries with query hints.

```sql
SELECT * FROM "lambda:snowflakeconnector".SYSTEM."MY_TABLE@schemaCase=upper&tableCase=upper"
```

```sql
SELECT * FROM "lambda:snowflakeconnector".SYSTEM."MY_TABLE@schemaCase=upper&tableCase=lower"
```

Terms

The following terms relate to the Snowflake connector.

• **Database instance** – Any instance of a database deployed on premises, on Amazon EC2, or on Amazon RDS.

• **Handler** – A Lambda handler that accesses your database instance. A handler can be for metadata or for data records.

• **Metadata handler** – A Lambda handler that retrieves metadata from your database instance.

• **Record handler** – A Lambda handler that retrieves data records from your database instance.

• **Composite handler** – A Lambda handler that retrieves both metadata and data records from your database instance.
• **Property or parameter** – A database property used by handlers to extract database information. You configure these properties as Lambda environment variables.

• **Connection String** – A string of text used to establish a connection to a database instance.

• **Catalog** – A non-AWS Glue catalog registered with Athena that is a required prefix for the connection_string property.

• **Multiplexing handler** – A Lambda handler that can accept and use multiple database connections.

### Parameters

Use the Lambda environment variables in this section to configure the Snowflake connector.

### Connection string

Use a JDBC connection string in the following format to connect to a database instance.

```
snowflake://${jdbc_connection_string}
```

### Using a multiplexing handler

You can use a multiplexer to connect to multiple database instances with a single Lambda function. Requests are routed by catalog name. Use the following classes in Lambda.

<table>
<thead>
<tr>
<th>Handler</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>SnowflakeMuxCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>SnowflakeMuxMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>SnowflakeMuxRecordHandler</td>
</tr>
</tbody>
</table>

### Multiplexing handler parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$catalog_connection_string</td>
<td>Required. A database instance connection string. Prefix the string with the name of the catalog used in Athena. For example, if the catalog registered with Athena is mysnowflakecatalog, then the environment variable name is mysnowflakecatalog_connection_string.</td>
</tr>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when the catalog is lambda:${AWS_LAMBDA_FUNCTION_NAME}.</td>
</tr>
</tbody>
</table>

The following example properties are for a Snowflake MUX Lambda function that supports two database instances: snowflake1 (the default), and snowflake2.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>snowflake://jdbc:snowflake://snowflake1.host:port/?warehouse=warehouse&amp;db=db1&amp;schema=schema1&amp;${Test/RDS/Snowflake1}</td>
</tr>
</tbody>
</table>
Providing credentials

To provide a user name and password for your database in your JDBC connection string, you can use connection string properties or AWS Secrets Manager.

- **Connection String** – A user name and password can be specified as properties in the JDBC connection string.
- **AWS Secrets Manager** – To use the Athena Federated Query feature with AWS Secrets Manager, the VPC connected to your Lambda function should have internet access or a VPC endpoint to connect to Secrets Manager.

You can put the name of a secret in AWS Secrets Manager in your JDBC connection string. The connector replaces the secret name with the username and password values from Secrets Manager.

For Amazon RDS database instances, this support is tightly integrated. If you use Amazon RDS, we highly recommend using AWS Secrets Manager and credential rotation. If your database does not use Amazon RDS, store the credentials as JSON in the following format:

```
{"username": ";(username)", "password": "$(password)"}
```

### Example connection string with secret name

The following string has the secret name ${Test/RDS/Snowflake1}.

```
snowflake://jdbc:snowflake://snowflake1.host:port/?
warehouse=warehousename&db=db1&schema=schema1${Test/RDS/Snowflake1}&...
```

The connector uses the secret name to retrieve secrets and provide the user name and password, as in the following example.

```
snowflake://jdbc:snowflake://snowflake1.host:port/
warehouse=warehousename&db=db1&schema=schema1&user=sample2&password=sample2&...
```

Currently, Snowflake recognizes the user and password JDBC properties. It also accepts the user name and password in the format `username/password` without the keys `user` or `password`.

### Using a single connection handler

You can use the following single connection metadata and record handlers to connect to a single Snowflake instance.

<table>
<thead>
<tr>
<th>Handler type</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>SnowflakeCompositeHandler</td>
</tr>
</tbody>
</table>
### Handler type

<table>
<thead>
<tr>
<th>Handler type</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metadata handler</td>
<td>SnowflakeMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>SnowflakeRecordHandler</td>
</tr>
</tbody>
</table>

### Single connection handler parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when a catalog is not recognized.</td>
</tr>
</tbody>
</table>

The single connection handlers support one database instance and must provide a `default` connection string parameter. All other connection strings are ignored.

The following example property is for a single Snowflake instance supported by a Lambda function.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>snowflake://jdbc:snowflake://snowflake1.host:port/?secret=Test/RDS/Snowflake1</td>
</tr>
</tbody>
</table>

### Spill parameters

The Lambda SDK can spill data to Amazon S3. All database instances accessed by the same Lambda function spill to the same location.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spill_bucket</td>
<td>Required. Spill bucket name.</td>
</tr>
<tr>
<td>spill_prefix</td>
<td>Required. Spill bucket key prefix.</td>
</tr>
<tr>
<td>spill_put_request_headers</td>
<td>(Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, <code>{&quot;x-amz-server-side-encryption&quot; : &quot;AES256&quot;}</code>). For other possible headers, see PutObject in the Amazon Simple Storage Service API Reference.</td>
</tr>
</tbody>
</table>

### Data type support

The following table shows the corresponding data types for JDBC and Apache Arrow.

<table>
<thead>
<tr>
<th>JDBC</th>
<th>Arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Bit</td>
</tr>
<tr>
<td>Integer</td>
<td>Tiny</td>
</tr>
<tr>
<td>Short</td>
<td>Smallint</td>
</tr>
</tbody>
</table>
In addition to the JDBC to Arrow conversions, the connector performs certain other conversions to make the Snowflake source and Athena data types compatible. These conversions help ensure that queries get executed successfully. The following table shows these conversions.

<table>
<thead>
<tr>
<th>Source data type (Snowflake)</th>
<th>Converted data type (Athena)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP</td>
<td>TIMESTAMPMILLI</td>
</tr>
<tr>
<td>DATE</td>
<td>TIMESTAMPMILLI</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INT</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>BIGINT</td>
</tr>
<tr>
<td>TIMESTAMP_NTZ</td>
<td>TIMESTAMPMILLI</td>
</tr>
</tbody>
</table>

All other unsupported data types are converted to VARCHAR.

**Partitions and splits**

Partitions are used to determine how to generate splits for the connector. Athena constructs a synthetic column of type VARCHAR that represents the partitioning scheme for the table to help the connector generate splits. The connector does not modify the actual table definition.

**Performance tuning**

For optimal performance, use filters in queries whenever possible. In addition, we highly recommend native partitioning to retrieve huge datasets that have uniform partition distribution.

**License information**

By using this connector, you acknowledge the inclusion of third party components, a list of which can be found in the pom.xml file for this connector, and agree to the terms in the respective third party licenses provided in the LICENSE.txt file on GitHub.com.
See also

For the latest JDBC driver version information, see the pom.xml file for the Snowflake connector on GitHub.com.

For additional information about this connector, visit the corresponding site on GitHub.com.

Amazon Athena Microsoft SQL Server connector

The Amazon Athena connector for Microsoft SQL Server enables Amazon Athena to run SQL queries on your data stored in Microsoft SQL Server using JDBC.

Prerequisites

- Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).
- Set up a VPC and a security group before you use this connector. For more information, see Creating a VPC for a data source connector (p. 66).

Limitations

- Write DDL operations are not supported.
- In a multiplexer setup, the spill bucket and prefix are shared across all database instances.
- Any relevant Lambda limits. For more information, see Lambda quotas in the AWS Lambda Developer Guide.
- In filter conditions, you must cast the Date and Timestamp data types to the appropriate data type.
- To search for negative values of type Real and Float, use the <= or >= operator.
- The binary, varbinary, image, and rowversion data types are not supported.

Terms

The following terms relate to the SQL Server connector.

- **Database instance** – Any instance of a database deployed on premises, on Amazon EC2, or on Amazon RDS.
- **Handler** – A Lambda handler that accesses your database instance. A handler can be for metadata or for data records.
- **Metadata handler** – A Lambda handler that retrieves metadata from your database instance.
- **Record handler** – A Lambda handler that retrieves data records from your database instance.
- **Composite handler** – A Lambda handler that retrieves both metadata and data records from your database instance.
- **Property or parameter** – A database property used by handlers to extract database information. You configure these properties as Lambda environment variables.
- **Connection String** – A string of text used to establish a connection to a database instance.
- **Catalog** – A non-AWS Glue catalog registered with Athena that is a required prefix for the connection_string property.
- **Multiplexing handler** – A Lambda handler that can accept and use multiple database connections.
Parameters

Use the Lambda environment variables in this section to configure the SQL Server connector.

Connection string

Use a JDBC connection string in the following format to connect to a database instance.

```
sqlserver://${jdbc_connection_string}
```

Using a multiplexing handler

You can use a multiplexer to connect to multiple database instances with a single Lambda function. Requests are routed by catalog name. Use the following classes in Lambda.

<table>
<thead>
<tr>
<th>Handler</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>SqlServerMuxCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>SqlServerMuxMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>SqlServerMuxRecordHandler</td>
</tr>
</tbody>
</table>

Multiplexing handler parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$catalog_connection_string</td>
<td>Required. A database instance connection string. Prefix the string with the name of the catalog used in Athena. For example, if the catalog registered with Athena is mysqlservercatalog, then the environment variable name is mysqlservercatalog_connection_string.</td>
</tr>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when the catalog is lambda:${AWS_LAMBDA_FUNCTION_NAME}.</td>
</tr>
</tbody>
</table>

The following example properties are for a SqlServer MUX Lambda function that supports two database instances: sqlserver1 (the default), and sqlserver2.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>sqlserver://jdbc:sqlserver://sqlserver1.hostname:port;databaseName=&lt;database_name&gt;;${secret1_name}</td>
</tr>
<tr>
<td>sqlserver_catalog1_connection</td>
<td>sqlserver://jdbc:sqlserver://sqlserver1.hostname:port;databaseName=&lt;database_name&gt;;${secret1_name}</td>
</tr>
<tr>
<td>sqlserver_catalog2_connection</td>
<td>sqlserver://jdbc:sqlserver://sqlserver2.hostname:port;databaseName=&lt;database_name&gt;;${secret2_name}</td>
</tr>
</tbody>
</table>
Providing credentials

To provide a user name and password for your database in your JDBC connection string, you can use connection string properties or AWS Secrets Manager.

- **Connection String** – A user name and password can be specified as properties in the JDBC connection string.
- **AWS Secrets Manager** – To use the Athena Federated Query feature with AWS Secrets Manager, the VPC connected to your Lambda function should have internet access or a VPC endpoint to connect to Secrets Manager.

You can put the name of a secret in AWS Secrets Manager in your JDBC connection string. The connector replaces the secret name with the username and password values from Secrets Manager.

For Amazon RDS database instances, this support is tightly integrated. If you use Amazon RDS, we highly recommend using AWS Secrets Manager and credential rotation. If your database does not use Amazon RDS, store the credentials as JSON in the following format:

```json
{"username": ":{username}", "password": ":{password}"}
```

**Example connection string with secret name**

The following string has the secret name `\{secret_name\}`.

```sqlserver://jdbc:sqlserver://hostname:port;databaseName=<database_name>;\{secret_name\}
```

The connector uses the secret name to retrieve secrets and provide the user name and password, as in the following example.

```sqlserver://jdbc:sqlserver://hostname:port;databaseName=<database_name>;user=<user>;password=<password>
```

**Using a single connection handler**

You can use the following single connection metadata and record handlers to connect to a single SQL Server instance.

<table>
<thead>
<tr>
<th>Handler type</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>SqlServerCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>SqlServerMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>SqlServerRecordHandler</td>
</tr>
</tbody>
</table>

**Single connection handler parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when a catalog is not recognized.</td>
</tr>
</tbody>
</table>

The single connection handlers support one database instance and must provide a default connection string parameter. All other connection strings are ignored.
The following example property is for a single SQL Server instance supported by a Lambda function.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>sqlserver://jdbc:sqlserver://hostname:port;databaseName=&lt;database_name&gt;;${secret_name}</td>
</tr>
</tbody>
</table>

**Spill parameters**

The Lambda SDK can spill data to Amazon S3. All database instances accessed by the same Lambda function spill to the same location.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spill_bucket</td>
<td>Required. Spill bucket name.</td>
</tr>
<tr>
<td>spill_prefix</td>
<td>Required. Spill bucket key prefix.</td>
</tr>
<tr>
<td>spill_put_request_headers</td>
<td>(Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, {&quot;x-amz-server-side-encryption&quot;: &quot;AES256&quot;}). For other possible headers, see PutObject in the Amazon Simple Storage Service API Reference.</td>
</tr>
</tbody>
</table>

**Data type support**

The following table shows the corresponding data types for SQL Server and Apache Arrow.

<table>
<thead>
<tr>
<th>SQL Server</th>
<th>Arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>TINYINT</td>
</tr>
<tr>
<td>tinyint</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>smallint</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>int</td>
<td>INT</td>
</tr>
<tr>
<td>bigint</td>
<td>BIGINT</td>
</tr>
<tr>
<td>decimal</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>numeric</td>
<td>FLOAT8</td>
</tr>
<tr>
<td>smallmoney</td>
<td>FLOAT8</td>
</tr>
<tr>
<td>money</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>float[24]</td>
<td>FLOAT4</td>
</tr>
<tr>
<td>float[53]</td>
<td>FLOAT8</td>
</tr>
<tr>
<td>real</td>
<td>FLOAT4</td>
</tr>
<tr>
<td>datetime</td>
<td>Date(MILLISECOND)</td>
</tr>
</tbody>
</table>
Partitions and splits

A partition is represented by a single partition column of type `varchar`. In case of the SQL Server connector, a partition function determines how partitions are applied on the table. The partition function and column name information are retrieved from the SQL Server metadata table. A custom query then gets the partition. Splits are created based upon the number of distinct partitions received.

License information

By using this connector, you acknowledge the inclusion of third party components, a list of which can be found in the `pom.xml` file for this connector, and agree to the terms in the respective third party licenses provided in the `LICENSE.txt` file on GitHub.com.

See also

For the latest JDBC driver version information, see the `pom.xml` file for the SQL Server connector on GitHub.com.

For additional information about this connector, visit the corresponding site on GitHub.com.

Amazon Athena Teradata connector

The Amazon Athena connector for Teradata enables Athena to run SQL queries on data stored in your Teradata databases.

Prerequisites

- Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).
- Set up a VPC and a security group before you use this connector. For more information, see Creating a VPC for a data source connector (p. 66).
Limitations

- Write DDL operations are not supported.
- In a multiplexer setup, the spill bucket and prefix are shared across all database instances.
- Any relevant Lambda limits. For more information, see Lambda quotas in the AWS Lambda Developer Guide.

Terms

The following terms relate to the Teradata connector.

- **Database instance** – Any instance of a database deployed on premises, on Amazon EC2, or on Amazon RDS.
- **Handler** – A Lambda handler that accesses your database instance. A handler can be for metadata or for data records.
- **Metadata handler** – A Lambda handler that retrieves metadata from your database instance.
- **Record handler** – A Lambda handler that retrieves data records from your database instance.
- **Composite handler** – A Lambda handler that retrieves both metadata and data records from your database instance.
- **Property or parameter** – A database property used by handlers to extract database information. You configure these properties as Lambda environment variables.
- **Connection String** – A string of text used to establish a connection to a database instance.
- **Catalog** – A non-AWS Glue catalog registered with Athena that is a required prefix for the connection_string property.
- **Multiplexing handler** – A Lambda handler that can accept and use multiple database connections.

Lambda layer prerequisite

To use the Teradata connector with Athena, you must create a Lambda layer that includes the Teradata JDBC driver. A Lambda layer is a .zip file archive that contains additional code for a Lambda function. When you deploy the Teradata connector to your account, you specify the layer's ARN. This attaches the Lambda layer with the Teradata JDBC driver to the Teradata connector so that you can use it with Athena.

For more information about Lambda layers, see Creating and sharing Lambda layers in the AWS Lambda Developer Guide.

To create a Lambda layer for the teradata connector

2. Download the Teradata JDBC driver. The website requires you to create an account and accept a license agreement to download the file.
3. Extract the terajdbc4.jar file from the archive file that you downloaded.
4. Create the following folder structure and place the .jar file in it.
   ```
   java\lib\terajdbc4.jar
   ```
5. Create a .zip file of the entire folder structure that contains the terajdbc4.jar file.
7. In the navigation pane, choose Layers, and then choose Create layer.
8. For Name, enter a name for the layer (for example, TeradataJava11LambdaLayer).
9. Ensure that the **Upload a .zip file** option is selected.
10. Choose **Upload**, and then upload the zipped folder that contains the Teradata JDBC driver.
11. Choose **Create**.
12. On the details page for the layer, copy the layer ARN by choosing the clipboard icon at the top of the page.
13. Save the ARN for reference.

**Parameters**

Use the Lambda environment variables in this section to configure the Teradata connector.

**Connection string**

Use a JDBC connection string in the following format to connect to a database instance.

```
teradata://${jdbc_connection_string}
```

**Using a multiplexing handler**

You can use a multiplexer to connect to multiple database instances with a single Lambda function. Requests are routed by catalog name. Use the following classes in Lambda.

<table>
<thead>
<tr>
<th>Handler</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>TeradataMuxCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>TeradataMuxMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>TeradataMuxRecordHandler</td>
</tr>
</tbody>
</table>

**Multiplexing handler parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$catalog_connection_string</td>
<td>Required. A database instance connection string. Prefix the string with the name of the catalog used in Athena. For example, if the catalog registered with Athena is myteradatasyncatalog, then the environment variable name is myteradatasyncatalog_connection_string.</td>
</tr>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when the catalog is lambda:${AWS_LAMBDA_FUNCTION_NAME}.</td>
</tr>
</tbody>
</table>

The following example properties are for a Teradata MUX Lambda function that supports two database instances: teradata1 (the default), and teradata2.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>teradata://jdbc:teradata://teradata2.host/TMODE=ANSI,CHARSET=UTF8,DATABASE=TEST,user=sample2&amp;password=sample2</td>
</tr>
</tbody>
</table>
### Property Values

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>teradata_catalog1_connection_string</td>
<td>teradata://jdbc:teradata://teradata1.host/TMODE=ANSI,CHARSET=UTF8,DATABASE=TEST,${Test/RDS/Teradata1}</td>
</tr>
<tr>
<td>teradata_catalog2_connection_string</td>
<td>teradata://jdbc:teradata://teradata2.host/TMODE=ANSI,CHARSET=UTF8,DATABASE=TEST,user=sample2&amp;password=sample2</td>
</tr>
</tbody>
</table>

### Providing credentials

To provide a user name and password for your database in your JDBC connection string, you can use connection string properties or AWS Secrets Manager.

- **Connection String** – A user name and password can be specified as properties in the JDBC connection string.
- **AWS Secrets Manager** – To use the Athena Federated Query feature with AWS Secrets Manager, the VPC connected to your Lambda function should have internet access or a VPC endpoint to connect to Secrets Manager.

You can put the name of a secret in AWS Secrets Manager in your JDBC connection string. The connector replaces the secret name with the username and password values from Secrets Manager.

For Amazon RDS database instances, this support is tightly integrated. If you use Amazon RDS, we highly recommend using AWS Secrets Manager and credential rotation. If your database does not use Amazon RDS, store the credentials as JSON in the following format:

```
{"username": "${username}", "password": "${password}"}
```

### Example connection string with secret name

The following string has the secret name `${Test/RDS/Teradata1}`.

```
teradata://jdbc:teradata1.host/TMODE=ANSI,CHARSET=UTF8,DATABASE=TEST,${Test/RDS/Teradata1}&...
```

The connector uses the secret name to retrieve secrets and provide the user name and password, as in the following example.

```
teradata://jdbc:teradata://teradata1.host/TMODE=ANSI,CHARSET=UTF8,DATABASE=TEST,...&user=sample2&password=sample2&...
```

Currently, Teradata recognizes the user and password JDBC properties. It also accepts the user name and password in the format `username/password` without the keys `user` or `password`.

### Using a single connection handler

You can use the following single connection metadata and record handlers to connect to a single Teradata instance.

<table>
<thead>
<tr>
<th>Handler type</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>TeradataCompositeHandler</td>
</tr>
<tr>
<td>Handler type</td>
<td>Class</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>TeradataMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>TeradataRecordHandler</td>
</tr>
</tbody>
</table>

**Single connection handler parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when a catalog is not recognized.</td>
</tr>
</tbody>
</table>

The single connection handlers support one database instance and must provide a `default` connection string parameter. All other connection strings are ignored.

The following example property is for a single Teradata instance supported by a Lambda function.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>teradata://jdbc:teradata://teradata1.host/TMODE=ANSI,CHARSET=UTF8,DATABASE=TEST,secret=Test/RDS/Teradata1</td>
</tr>
</tbody>
</table>

**Spill parameters**

The Lambda SDK can spill data to Amazon S3. All database instances accessed by the same Lambda function spill to the same location.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spill_bucket</td>
<td>Required. Spill bucket name.</td>
</tr>
<tr>
<td>spill_prefix</td>
<td>Required. Spill bucket key prefix.</td>
</tr>
<tr>
<td>spill_put_request_headers</td>
<td>(Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, {&quot;x-amz-server-side-encryption&quot; : &quot;AES256&quot;}). For other possible headers, see PutObject in the Amazon Simple Storage Service API Reference.</td>
</tr>
</tbody>
</table>

**Data type support**

The following table shows the corresponding data types for JDBC and Apache Arrow.

<table>
<thead>
<tr>
<th>JDBC</th>
<th>Arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Bit</td>
</tr>
<tr>
<td>Integer</td>
<td>Tiny</td>
</tr>
<tr>
<td>JDBC</td>
<td>Arrow</td>
</tr>
<tr>
<td>------------</td>
<td>--------</td>
</tr>
<tr>
<td>Short</td>
<td>Smallint</td>
</tr>
<tr>
<td>Integer</td>
<td>Int</td>
</tr>
<tr>
<td>Long</td>
<td>Bigint</td>
</tr>
<tr>
<td>float</td>
<td>Float4</td>
</tr>
<tr>
<td>Double</td>
<td>Float8</td>
</tr>
<tr>
<td>Date</td>
<td>DateDay</td>
</tr>
<tr>
<td>Timestamp</td>
<td>DateMilli</td>
</tr>
<tr>
<td>String</td>
<td>Varchar</td>
</tr>
<tr>
<td>Bytes</td>
<td>Varbinary</td>
</tr>
<tr>
<td>BigDecimal</td>
<td>Decimal</td>
</tr>
<tr>
<td>ARRAY</td>
<td>List</td>
</tr>
</tbody>
</table>

Partitions and splits

A partition is represented by a single partition column of type Integer. The column contains partition names of the partitions defined on a Teradata table. For a table that does not have partition names, * is returned, which is equivalent to a single partition. A partition is equivalent to a split.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>partition</td>
<td>Integer</td>
<td>Named partition in Teradata.</td>
</tr>
</tbody>
</table>

Performance tuning

Teradata supports native partitions. The Athena Lambda connector can retrieve data from these partitions in parallel. If you want to query very large datasets with uniform partition distribution, native partitioning is highly recommended.

License information

By using this connector, you acknowledge the inclusion of third party components, a list of which can be found in the pom.xml file for this connector, and agree to the terms in the respective third party licenses provided in the LICENSE.txt file on GitHub.com.

See also

For the latest JDBC driver version information, see the pom.xml file for the Teradata connector on GitHub.com.

For additional information about this connector, visit the corresponding site on GitHub.com.

Amazon Athena Timestream connector

The Amazon Athena Timestream connector enables Amazon Athena to communicate with Amazon Timestream, making your time series data accessible through Amazon Athena. You can optionally use AWS Glue Data Catalog as a source of supplemental metadata.
Amazon Timestream is a fast, scalable, fully managed, purpose-built time series database that makes it easy to store and analyze trillions of time series data points per day. Timestream saves you time and cost in managing the lifecycle of time series data by keeping recent data in memory and moving historical data to a cost optimized storage tier based upon user defined policies.

Prerequisites

• Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).

Parameters

Use the Lambda environment variables in this section to configure the Timestream connector.

• **spill_bucket** – Specifies the Amazon S3 bucket for data that exceeds Lambda function limits.
• **spill_prefix** – (Optional) Defaults to a subfolder in the specified spill_bucket called athena-federation-spill. We recommend that you configure an Amazon S3 storage lifecycle on this location to delete spills older than a predetermined number of days or hours.
• **spill_put_request_headers** – (Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, \{"x-amz-server-side-encryption" : "AES256\"\}). For other possible headers, see PutObject in the Amazon Simple Storage Service API Reference.
• **kms_key_id** – (Optional) By default, any data that is spilled to Amazon S3 is encrypted using the AES-GCM authenticated encryption mode and a randomly generated key. To have your Lambda function use stronger encryption keys generated by KMS like a7e63k4b-81oc-40db-a2a1-4d0en2cd8331, you can specify a KMS key ID.
• **disable_spill_encryption** – (Optional) When set to True, disables spill encryption. Defaults to False so that data that is spilled to S3 is encrypted using AES-GCM – either using a randomly generated key or KMS to generate keys. Disabling spill encryption can improve performance, especially if your spill location uses server-side encryption.
• **glue_catalog** – (Optional) Use this option to specify a cross-account AWS Glue catalog (p. 24). By default, the connector attempts to get metadata from its own AWS Glue account.

Setting up databases and tables in AWS Glue

You can optionally use the AWS Glue Data Catalog as a source of supplemental metadata. To enable an AWS Glue table for use with Timestream, you must have an AWS Glue database and table with names that match the Timestream database and table that you want to supply supplemental metadata for. To configure AWS Glue table for use with Timestream, you must set its table properties in AWS Glue.

To use an AWS Glue table for supplemental metadata

1. Edit the table in the AWS Glue console to add the following table properties:
   - **timestream-metadata-flag** – This property indicates to the Timestream connector that the connector can use the table for supplemental metadata. You can provide any value for timestream-metadata-flag as long as the timestream-metadata-flag property is present in the list of table properties.
   - **_view_template** – When you use AWS Glue for supplemental metadata, you can use this table property and specify any Timestream SQL as the view. The Athena Timestream connector uses the SQL from the view together with your SQL from Athena to run your query. This is useful if you want to use a feature of Timestream SQL that is not otherwise available in Athena.
2. Make sure that you use the data types appropriate for AWS Glue as listed in this document.
Data types

Currently, the Timestream connector supports only a subset of the data types available in Timestream, specifically: the scalar values varchar, double, and timestamp.

To query the timeseries data type, you must configure a view in AWS Glue table properties that uses the Timestream CREATE_TIME_SERIES function. You also need to provide a schema for the view that uses the syntax ARRAY<STRUCT<time:timestamp, measure_value:double>> as the type for any of your time series columns. Be sure to replace double with the appropriate scalar type for your table.

The following image shows an example of AWS Glue table properties configured to set up a view over a time series.

![AWS Glue Table Properties Example](image)

Required Permissions

Review the Policies section of the athena-timestream.yaml file for full details on the IAM policies that this connector requires. The following is a brief summary.

- **Amazon S3 write access** – To successfully handle large queries, the connector requires write access to a location in Amazon S3.
- **Athena GetQueryExecution** – The connector uses this permission to fast-fail when the upstream Athena query has terminated.
- **AWS Glue Data Catalog** – The Timestream connector requires read only access to the AWS Glue Data Catalog to obtain schema information.
- **CloudWatch Logs** – The connector requires access to CloudWatch Logs for storing logs.
- **Timestream Access** – For running Timestream queries.

Performance

Currently, queries that run using this connector are significantly slower (more than twice as slow) than queries that run from Timestream itself. We recommend that you limit the data returned (not the data
scanned) to less than 256MB. Note that unique and interesting use cases are possible even well below the 256MB recommendation.

License information

The Amazon Athena Timestream connector project is licensed under the Apache-2.0 License.

See also

For additional information about this connector, visit the corresponding site on GitHub.com.

Amazon Athena TPC benchmark DS (TPC-DS) connector

The Amazon Athena TPC-DS connector enables Amazon Athena to communicate with a source of randomly generated TPC Benchmark DS data for use in benchmarking and functional testing of Athena Federation. The Athena TPC-DS connector generates a TPC-DS compliant database at one of four scale factors. We do not recommend the use of this connector as an alternative to Amazon S3-based data lake performance tests.

Prerequisites

- Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).

Parameters

Use the Lambda environment variables in this section to configure the TPC-DS connector.

- **spill_bucket** – Specifies the Amazon S3 bucket for data that exceeds Lambda function limits.
- **spill_prefix** – (Optional) Defaults to a subfolder in the specified spill_bucket called athena-federation-spill. We recommend that you configure an Amazon S3 storage lifecycle on this location to delete spills older than a predetermined number of days or hours.
- **spill_put_request_headers** – (Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, {"x-amz-server-side-encryption" : "AES256"}). For other possible headers, see PutObject in the Amazon Simple Storage Service API Reference.
- **kms_key_id** – (Optional) By default, any data that is spilled to Amazon S3 is encrypted using the AES-GCM authenticated encryption mode and a randomly generated key. To have your Lambda function use stronger encryption keys generated by KMS like a7e63k4b-8loc-40db-a2a1-4d0en2cd8331, you can specify a KMS key ID.
- **disable_spill_encryption** – (Optional) When set to True, disables spill encryption. Defaults to False so that data that is spilled to S3 is encrypted using AES-GCM – either using a randomly generated key or KMS to generate keys. Disabling spill encryption can improve performance, especially if your spill location uses server-side encryption.

Test databases and tables

The Athena TPC-DS connector generates a TPC-DS compliant database at one of the four scale factors tpcds1, tpcds10, tpcds100, tpcds250, or tpcds1000.

Summary of tables

For a complete list of the test data tables and columns, run the SHOW TABLES or DESCRIBE TABLE queries. The following summary of tables is provided for convenience.
1. call_center  
2. catalog_page  
3. catalog_returns  
4. catalog_sales  
5. customer  
6. customer_address  
7. customer_demographics  
8. date_dim  
9. dbgen_version  
10. household_demographics  
11. income_band  
12. inventory  
13. item  
14. promotion  
15. reason  
16. ship_mode  
17. store  
18. store_returns  
19. store_sales  
20. time_dim  
21. warehouse  
22. web_page  
23. web_returns  
24. web_sales  
25. web_site  

For TPC-DS queries that are compatible with this generated schema and data, see the athena-tpcids/src/main/resources/queries/ directory on GitHub.

**Example query**

The following SELECT query example queries the tpcds catalog for customer demographics in specific counties.

```
SELECT  
  cd_gender,  
  cd_marital_status,  
  cd_education_status,  
  count(*) cnt1,  
  cd_purchase_estimate,  
  count(*) cnt2,  
  cd_credit_rating,  
  count(*) cnt3,  
  cd_dep_count,  
  count(*) cnt4,  
  cd_dep_employed_count,  
  count(*) cnt5,  
  cd_dep_college_count,  
  count(*) cnt6  
FROM  
```
"lambda:tpcds".tpcdsl.customer c, "lambda:tpcds".tpcdsl.customer_address ca, "lambda:tpcds".tpcdsl.customer_demographics
WHERE
c.c_current_addr_sk = ca.ca_address_sk AND
ca_county IN ('Rush County', 'Toole County', 'Jefferson County', 'Dona Ana County', 'La Porte County') AND
cd_demo_sk = c.c_current_cdemo_sk AND
exists(SELECT *
FROM "lambda:tpcds".tpcdsl.store_sales, "lambda:tpcds".tpcdsl.date_dim
WHERE c.c_customer_sk = ss_customer_sk AND
ss_sold_date_sk = d_date_sk AND
d_year = 2002 AND
d_moy BETWEEN 1 AND 1 + 3) AND
(exists(SELECT *
FROM "lambda:tpcds".tpcdsl.web_sales, "lambda:tpcds".tpcdsl.date_dim
WHERE c.c_customer_sk = ws_bill_customer_sk AND
ws_sold_date_sk = d_date_sk AND
d_year = 2002 AND
d_moy BETWEEN 1 AND 1 + 3) OR
exists(SELECT *
FROM "lambda:tpcds".tpcdsl.catalog_sales, "lambda:tpcdsl".tpcdsl.date_dim
WHERE c.c_customer_sk = cs_ship_customer_sk AND
cs_sold_date_sk = d_date_sk AND
d_year = 2002 AND
d_moy BETWEEN 1 AND 1 + 3))
GROUP BY cd_gender,
cd_marital_status,
cd_education_status,
cd_purchase_estimate,
cd_credit_rating,
cd_dep_count,
cd_dep_employed_count,
cd_dep_college_count
ORDER BY cd_gender,
cd_marital_status,
cd_education_status,
cd_purchase_estimate,
cd_credit_rating,
cd_dep_count,
cd_dep_employed_count,
cd_dep_college_count
LIMIT 100

Required Permissions

Review the Policies section of the athena-tpcds.yaml file for full details on the IAM policies that this connector requires. The following is a brief summary.

- **Amazon S3 write access** – To successfully handle large queries, the connector requires write access to a location in Amazon S3.
- **Athena GetQueryExecution** – The connector uses this permission to fast-fail when the upstream Athena query has terminated.

Performance

The Athena TPC-DS connector attempts to parallelize queries based on the scale factor that you choose. Predicate pushdown is performed within the Lambda function.

License information

The Amazon Athena TPC-DS connector project is licensed under the Apache-2.0 License.
See also

For additional information about this connector, visit the corresponding site on GitHub.com.

Amazon Athena Vertica connector

Vertica is a columnar database platform that can be deployed in the cloud or on premises that supports exabyte scale data warehouses. You can use the Amazon Athena Vertica connector in federated queries to query Vertica data sources from Athena. For example, you can run analytical queries over a data warehouse on Vertica and a data lake in Amazon S3.

Prerequisites

• Deploy the connector to your AWS account using the Athena console or the AWS Serverless Application Repository. For more information, see Deploying a connector and connecting to a data source (p. 62) or Using the AWS Serverless Application Repository to deploy a data source connector (p. 64).

• Set up a VPC and a security group before you use this connector. For more information, see Creating a VPC for a data source connector (p. 66).

Limitations

• Because the Athena Vertica connector uses Amazon S3 Select to read Parquet files from Amazon S3, performance of the connector can be slow. When you query large tables, we recommend that you use a CREATE TABLE AS (SELECT ...) (p. 281) query and SQL predicates.

• Currently, due to a known issue in Athena Federated Query, the connector causes Vertica to export all columns of the queried table to Amazon S3, but only the queried columns are visible in the results on the Athena console.

• Write DDL operations are not supported.

• Any relevant Lambda limits. For more information, see Lambda quotas in the AWS Lambda Developer Guide.

Workflow

The following diagram shows the workflow of a query that uses the Vertica connector.

1. A SQL query is issued against one or more tables in Vertica.
2. The connector parses the SQL query to send the relevant portion to Vertica through the JDBC connection.
3. The connection strings use the user name and password stored in AWS Secrets Manager to gain access to Vertica.
4. The connector wraps the SQL query with a Vertica EXPORT command, as in the following example.

```sql
EXPORT TO PARQUET (directory = 's3://bucket_name/folder_name',
    Compression='Snappy', fileSizeMB=64) OVER() as
SELECT
    PATH_ID,
...
SOURCE_ITEMIZED,
SOURCE_OVERRIDE
FROM DELETED_OBJECT_SCHEMA.FORM_USAGE_DATA
WHERE PATH_ID <= 5;
```
5. Vertica processes the SQL query and sends the result set to an Amazon S3 bucket. For better throughput, Vertica uses the EXPORT option to parallelize the write operation of multiple Parquet files.
6. Athena scans the Amazon S3 bucket to determine the number of files to read for the result set.
7. Athena makes multiple calls to the Lambda function and uses Amazon S3 Select to read the Parquet files from the result set. Multiple calls enable Athena to parallelize the read of the Amazon S3 files and achieve a throughput of up to 100GB per second.
8. Athena processes the data returned from Vertica with data scanned from the data lake and returns the result.

**Terms**

The following terms relate to the Vertica connector.

- **Database instance** – Any instance of a Vertica database deployed on Amazon EC2.
- **Handler** – A Lambda handler that accesses your database instance. A handler can be for metadata or for data records.
- **Metadata handler** – A Lambda handler that retrieves metadata from your database instance.
- **Record handler** – A Lambda handler that retrieves data records from your database instance.
- **Composite handler** – A Lambda handler that retrieves both metadata and data records from your database instance.
- **Property or parameter** – A database property used by handlers to extract database information. You configure these properties as Lambda environment variables.
- **Connection String** – A string of text used to establish a connection to a database instance.
- **Catalog** – A non-AWS Glue catalog registered with Athena that is a required prefix for the connection_string property.

**Parameters**

The Amazon Athena Vertica connector exposes configuration options through Lambda environment variables. You can use the following Lambda environment variables to configure the connector.

- **AthenaCatalogName** – Lambda function name
- **ExportBucket** – The Amazon S3 bucket where the Vertica query results are exported.
- **SpillBucket** – The name of the Amazon S3 bucket where this function can spill data.
- **SpillPrefix** – The prefix for the SpillBucket location where this function can spill data.
- **SecurityGroupIds** – One or more IDs that correspond to the security group that should be applied to the Lambda function (for example, sg1, sg2, or sg3).
- **SubnetIds** – One or more subnet IDs that correspond to the subnet that the Lambda function can use to access your data source (for example, subnet1, or subnet2).
- **SecretNameOrPrefix** – The name or prefix of a set of names in Secrets Manager that this function has access to (for example, vertica-*).
- **VerticaConnectionString** – The Vertica connection details to use by default if no catalog specific connection is defined. The string can optionally use AWS Secrets Manager syntax (for example, ${secret_name}).
- **VPC ID** – The VPC ID to be attached to the Lambda function.

### Connection string

Use a JDBC connection string in the following format to connect to a database instance.

```java
jdbc:vertica://host_name:port/database?user=vertica-username&password=vertica-password
```

#### Using a single connection handler

You can use the following single connection metadata and record handlers to connect to a single Vertica instance.

<table>
<thead>
<tr>
<th>Handler type</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite handler</td>
<td>VerticaCompositeHandler</td>
</tr>
<tr>
<td>Metadata handler</td>
<td>VerticaMetadataHandler</td>
</tr>
<tr>
<td>Record handler</td>
<td>VerticaRecordHandler</td>
</tr>
</tbody>
</table>

### Single connection handler parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Required. The default connection string. This string is used when a catalog is not recognized.</td>
</tr>
</tbody>
</table>

The single connection handlers support one database instance and must provide a default connection string parameter. All other connection strings are ignored.

#### Providing credentials

To provide a user name and password for your database in your JDBC connection string, you can use connection string properties or AWS Secrets Manager.

- **Connection String** – A user name and password can be specified as properties in the JDBC connection string.
- **AWS Secrets Manager** – To use the Athena Federated Query feature with AWS Secrets Manager, the VPC connected to your Lambda function should have internet access or a VPC endpoint to connect to Secrets Manager.
You can put the name of a secret in AWS Secrets Manager in your JDBC connection string. The connector replaces the secret name with the `username` and `password` values from Secrets Manager.

For Amazon RDS database instances, this support is tightly integrated. If you use Amazon RDS, we highly recommend using AWS Secrets Manager and credential rotation. If your database does not use Amazon RDS, store the credentials as JSON in the following format:

```
{"username": ";{username}" , "password" : "${password}"}
```

**Example connection string with secret names**

The following string has the secret names `${vertica-username}` and `${vertica-password}`.

```
jdbc:vertica://host_name:port/database?user=${vertica-username}&password=${vertica-password}
```

The connector uses the secret name to retrieve secrets and provide the user name and password, as in the following example.

```
jdbc:vertica://host_name:port/database?user=sample-user&password=sample-password
```

Currently, the Vertica connector recognizes the `vertica-username` and `vertica-password` JDBC properties.

**Spill parameters**

The Lambda SDK can spill data to Amazon S3. All database instances accessed by the same Lambda function spill to the same location.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spill_bucket</td>
<td>Required. Spill bucket name.</td>
</tr>
<tr>
<td>spill_prefix</td>
<td>Required. Spill bucket key prefix.</td>
</tr>
<tr>
<td>spill_put_request_headers</td>
<td>(Optional) A JSON encoded map of request headers and values for the Amazon S3 putObject request that is used for spilling (for example, <code>{&quot;x-amz-server-side-encryption&quot; : &quot;AES256&quot;}</code>). For other possible headers, see PutObject in the Amazon Simple Storage Service API Reference.</td>
</tr>
</tbody>
</table>

**Data type support**

The following table shows the supported data types for the Vertica connector.

<table>
<thead>
<tr>
<th>Boolean</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BigInt</td>
<td>Short</td>
</tr>
<tr>
<td>Integer</td>
<td></td>
</tr>
</tbody>
</table>
Writing federated queries

After you have configured one or more data connectors and deployed them to your account, you can use them in your Athena queries.

Querying a single data source

The examples in this section assume that you have configured and deployed the Athena CloudWatch connector to your account. Use the same approach to query when you use other connectors.

To create an Athena query that uses the CloudWatch connector

2. In the Athena query editor, create a SQL query that uses the following syntax in the FROM clause.

```sql
MyCloudwatchCatalog.database_name.table_name
```

Examples

The following example uses the Athena CloudWatch connector to connect to the all_log_streams view in the /var/ecommerce-engine/order-processor CloudWatch Logs Log group. The
all_log_streams view is a view of all the log streams in the log group. The example query limits the number of rows returned to 100.

**Example**

```sql
SELECT * FROM "MyCloudwatchCatalog"./var/ecommerce-engine/order-processor".all_log_streams
limit 100;
```

The following example parses information from the same view as the previous example. The example extracts the order ID and log level and filters out any message that has the level INFO.

**Example**

```sql
SELECT log_stream as ec2_instance,
    Regexp_extract(message '.*orderId=(\d+) .*', 1) AS orderId,
    message AS order_processor_log,
    Regexp_extract(message, '(.*):.*', 1) AS log_level
FROM "MyCloudwatchCatalog"./var/ecommerce-engine/order-processor".all_log_streams
WHERE Regexp_extract(message, '(.*):.*', 1) != 'INFO'
```

The following image shows a sample result.

**Note**
This example shows a query where the data source has been registered as a catalog with Athena. You can also reference a data source connector Lambda function using the format `lambda:MyLambdaFunctionName`.

**Querying multiple data sources**

As a more complex example, imagine an ecommerce company that has an application infrastructure such as the one shown in the following diagram.
The following descriptions explain the numbered items in the diagram.

1. Payment processing in a secure VPC with transaction records stored in HBase on Amazon EMR
2. Redis to store active orders so that the processing engine can access them quickly
3. Amazon DocumentDB for customer account data such as email addresses and shipping addresses
4. A product catalog in Amazon Aurora for an ecommerce site that uses automatic scaling on Fargate
5. CloudWatch Logs to house the order processor’s log events
6. A write-once-read-many data warehouse on Amazon RDS
7. DynamoDB to store shipment tracking data

Imagine that a data analyst for this ecommerce application discovers that the state of some orders is being reported erroneously. Some orders show as pending even though they were delivered, while others show as delivered but haven’t shipped.

The analyst wants to know how many orders are being delayed and what the affected orders have in common across the ecommerce infrastructure. Instead of investigating the sources of information separately, the analyst federates the data sources and retrieves the necessary information in a single query. Extracting the data into a single location is not necessary.

The analyst’s query uses the following Athena data connectors:

- **CloudWatch Logs** – Retrieves logs from the order processing service and uses regex matching and extraction to filter for orders with WARN or ERROR events.
- **Redis** – Retrieves the active orders from the Redis instance.
- **CMDB** – Retrieves the ID and state of the Amazon EC2 instance that ran the order processing service and logged the WARN or ERROR message.
• **DocumentDB** – Retrieves the customer email and address from Amazon DocumentDB for the affected orders.

• **DynamoDB** – Retrieves the shipping status and tracking details from the shipping table to identify possible discrepancies between reported and actual status.

• **HBase** – Retrieves the payment status for the affected orders from the payment processing service.

### Example

**Note**

This example shows a query where the data source has been registered as a catalog with Athena. You can also reference a data source connector Lambda function using the format `lambda:MyLambdaFunctionName`.

```sql
-- Sample query using multiple Athena data connectors.
WITH logs AS (SELECT log_stream, message AS order_processor_log, Regexp_extract(message, '.*orderId=(\d+) .*', 1) AS orderId, Regexp_extract(message, '(.*):.*', 1) AS log_level
FROM "MyCloudwatchCatalog"."/var/eCommerce-engine/order-processor".all_log_streams
WHERE Regexp_extract(message, '(.*):.*', 1) != 'INFO'),
active_orders AS (SELECT * FROM redis.redis_db.redis_customer_orders),
order_processors AS (SELECT instanceid, publicipaddress, state.NAME
FROM awscmdb.ec2.ec2_instances),
customer AS (SELECT id, email FROM docdb.customers.customer_info),
addresses AS (SELECT id, is_residential, address.street AS street
FROM docdb.customers.customer_addresses),
shipments AS (SELECT order_id, shipment_id, from_unixtime(cast(shipped_date as double)) as shipment_time, carrier
FROM lambda_ddb.default.order_shipments),
FROM "hbase".hbase_payments.transactions)
SELECT _key_ AS redis_order_id, customer_id, customer.email AS cust_email, "summary:cc_id" AS credit_card, "details:network" AS CC_type, "summary:status" AS payment_status, status AS redis_status, addresses.street AS street_address, shipments.shipment_time as shipment_time, shipments.carrier as shipment_carrier,
```
Enabling cross-account federated queries

Federated query allows you to query data sources other than Amazon S3 using data source connectors deployed on AWS Lambda. The cross-account federated query feature allows the Lambda function and the data sources that are to be queried to be located in different accounts.

As a data administrator, you can enable cross-account federated queries by sharing your data connector with a data analyst's account or, as a data analyst, by using a shared Lambda ARN from a data administrator to add to your account. When configuration changes are made to a connector in the originating account, the updated configuration is automatically applied to the shared instances of the connector in other user's accounts.

Considerations and limitations

- The cross-account federated query feature is available for non-Hive metastore data connectors that use a Lambda-based data source.
- The feature is not available for the AWS Glue Data Catalog data source type. For information about cross-account access to AWS Glue Data Catalogs, see Cross-account access to AWS Glue data catalogs (p. 548).

Required permissions

- For data administrator Account A to share a Lambda function with data analyst Account B, Account B requires Lambda invoke function and spill bucket access. Accordingly, Account A should add a resource-based policy to the Lambda function and principal access to its spill bucket in Amazon S3.

1. The following policy grants Lambda invoke function permissions to Account B on a Lambda function in Account A.
Enabling cross-account federated queries

2. The following policy allows spill bucket access to the principal in Account B.

```json
{
  "Version": "2008-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": ["arn:aws:iam::account-B-id:user/username"]
      },
      "Action": ["s3:GetObject", "s3:ListBucket"],
      "Resource": ["arn:aws:s3:::spill-bucket", "arn:aws:s3:::spill-bucket/*"]
    }
  ]
}
```

3. If the Lambda function is encrypting the spill bucket with an AWS KMS key instead of the default encryption offered by the federation SDK, the AWS KMS key policy in Account A must grant access to the user in Account B, as in the following example.

```json
{
  "Sid": "Allow use of the key",
  "Effect": "Allow",
  "Principal": {
    "AWS": ["arn:aws:iam::account-B-id:user/username"]
  },
  "Action": [ "kms:Decrypt" ],
  "Resource": [ "*" ] // Resource policy that gets placed on the KMS key.
}
```

- For Account A to share its connector with Account B, Account B must create a role called `AthenaCrossAccountCreate-account-A-id` that Account A assumes by calling the AWS Security Token Service `AssumeRole` API action.

The following policy, which allows the `CreateDataCatalog` action, should be created in Account B and added to the `AthenaCrossAccountCreate-account-A-id` role that Account B creates for Account A.
Sharing a data source in Account A with Account B

After permissions are in place, you can use the **Data sources** page in the Athena console to share a data connector in your account (Account A) with another account (Account B). Account A retains full control and ownership of the connector. When Account A makes configuration changes to the connector, the updated configuration applies to the shared connector in Account B.

**To share a Lambda data source in Account A with Account B**

2. If the console navigation pane is not visible, choose the expansion menu on the left.
3. Choose **Data sources**.
4. On the **Data sources** page, choose the link of the connector that you want to share.
5. On the details page for a Lambda data source, choose the **Share** option in the top right corner.
6. In the **Share Lambda-name with another account** dialog box, enter the required information.

   - For **Data source name**, enter the name of the copied data source as you want it to appear in the other account.
   - For **Account ID**, enter the ID of the account with which you want to share your data source (in this case, Account B).
Choose Share. The shared data connector that you specified is created in Account B. Configuration changes to the connector in Account A apply to the connector in Account B.

**Adding a shared data source from Account A to Account B**

As a data analyst, you may be given the ARN of a connector to add to your account from a data administrator. You can use the Data sources page of the Athena console to add the Lambda ARN provided by your administrator to your account.

**To add the Lambda ARN of a shared data connector to your account**

2. If you are using the new console experience and the navigation pane is not visible, choose the expansion menu on the left.
3. Choose Data sources.
4. On the Data sources page, choose Connect data source.
5. Choose **Custom or shared connector**.

6. In the **Lambda function** section, make sure that the option **Use an existing Lambda function** is selected.
7. For **Choose or enter a Lambda function**, enter the Lambda ARN of Account A.

8. Choose **Connect data source**.

**Troubleshooting**

If you receive an error message that Account A does not have the permissions to assume a role in Account B, make sure that the name of the role created in Account B is spelled correctly and that it has the proper policy attached.

**Writing a data source connector using the Athena Query Federation SDK**

To write your own **data source connectors** (p. 60), you can use the **Athena Query Federation SDK**. The Athena Query Federation SDK defines a set of interfaces and wire protocols that you can use to enable Athena to delegate portions of its query execution plan to code that you write and deploy. The SDK includes a connector suite and an example connector.

You can also customize Amazon Athena’s **prebuilt connectors** for your own use. You can modify a copy of the source code from GitHub and then use the **Connector publish tool** to create your own AWS Serverless Application Repository package. After you deploy your connector in this way, you can use it in your Athena queries.
IAM policies for accessing data catalogs

To control access to data catalogs, use resource-level IAM permissions or identity-based IAM policies.

The following procedure is specific to Athena.

For IAM-specific information, see the links listed at the end of this section. For information about example JSON data catalog policies, see Data Catalog example policies (p. 179).

To use the visual editor in the IAM console to create a data catalog policy

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane on the left, choose Policies, and then choose Create policy.
3. On the Visual editor tab, choose Choose a service. Then choose Athena to add to the policy.
4. Choose Select actions, and then choose the actions to add to the policy. The visual editor shows the actions available in Athena. For more information, see Actions, resources, and condition keys for Amazon Athena in the Service Authorization Reference.
5. Choose add actions to type a specific action or use wildcards (*) to specify multiple actions.

By default, the policy that you are creating allows the actions that you choose. If you chose one or more actions that support resource-level permissions to the datacatalog resource in Athena, then the editor lists the datacatalog resource.

6. Choose Resources to specify the specific data catalogs for your policy. For example JSON data catalog policies, see Data Catalog example policies (p. 179).
7. Specify the datacatalog resource as follows:

```
arn:aws:athena:<region>:<user-account>:datacatalog/<datacatalog-name>
```

8. Choose Review policy, and then type a Name and a Description (optional) for the policy that you are creating. Review the policy summary to make sure that you granted the intended permissions.
9. Choose Create policy to save your new policy.
10. Attach this identity-based policy to a user, a group, or role and specify the datacatalog resources they can access.

For more information, see the following topics in the Service Authorization Reference and the IAM User Guide:

- Actions, resources, and condition keys for Amazon Athena
- Creating policies with the visual editor
- Adding and removing IAM policies
- Controlling access to resources

For example JSON data catalog policies, see Data Catalog example policies (p. 179).
For information about AWS Glue permissions and AWS Glue crawler permissions, see Setting up IAM permissions for AWS Glue and Crawler prerequisites in the AWS Glue Developer Guide.

For a complete list of Amazon Athena actions, see the API action names in the Amazon Athena API Reference.

## Data Catalog example policies

This section includes example policies you can use to enable various actions on data catalogs.

A data catalog is an IAM resource managed by Athena. Therefore, if your data catalog policy uses actions that take datacatalog as an input, you must specify the data catalog's ARN as follows:

```
```

The <datacatalog-name> is the name of your data catalog. For example, for a data catalog named test_datacatalog, specify it as a resource as follows:

```
"Resource": ["arn:aws:athena:us-east-1:123456789012:datacatalog/test_datacatalog"]
```

For a complete list of Amazon Athena actions, see the API action names in the Amazon Athena API Reference. For more information about IAM policies, see Creating policies with the visual editor in the IAM User Guide. For more information about creating IAM policies for workgroups, see IAM policies for accessing data catalogs (p. 178).

- Example Policy for Full Access to All Data Catalogs (p. 179)
- Example Policy for Full Access to a Specified Data Catalog (p. 179)
- Example Policy for Querying a Specified Data Catalog (p. 180)
- Example Policy for Management Operations on a Specified Data Catalog (p. 181)
- Example Policy for Listing Data Catalogs (p. 181)
- Example Policy for Metadata Operations on Data Catalogs (p. 182)

### Example Example policy for full access to all data catalogs

The following policy allows full access to all data catalog resources that might exist in the account. We recommend that you use this policy for those users in your account that must administer and manage data catalogs for all other users.

```json
{
  "Version":"2012-10-17",
  "Statement":[
    {
      "Effect":"Allow",
      "Action":[
        "athena:*"
      ],
      "Resource":[
        "*"
      ]
    }
  ]
}
```

### Example Example policy for full access to a specified Data Catalog

The following policy allows full access to the single specific data catalog resource, named datacatalogA. You could use this policy for users with full control over a particular data catalog.

```json
{
  "Version":"2012-10-17",
  "Statement":[
    {
      "Effect":"Allow",
      "Action":[
        "athena:*"
      ],
      "Resource":[
        "arn:aws:athena:us-east-1:123456789012:datacatalog/test_datacatalog"
      ]
    }
  ]
}
```
Example Example policy for querying a specified Data Catalog

In the following policy, a user is allowed to run queries on the specified datacatalogA. The user is not allowed to perform management tasks for the data catalog itself, such as updating or deleting it.

```json
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect":"Allow",
            "Action": [
                "athena:ListDataCatalogs",
                "athena:ListWorkGroups",
                "athena:GetDatabase",
                "athena:ListDatabases",
                "athena:ListTableMetadata",
                "athena:GetTableMetadata"
            ],
            "Resource": "*"
        },
        {
            "Effect":"Allow",
            "Action": [
                "athena:StartQueryExecution",
                "athena:GetQueryResults",
                "athena:DeleteNamedQuery",
                "athena:GetNamedQuery",
                "athena:ListQueryExecutions",
                "athena:StopQueryExecution",
                "athena:GetQueryResultsStream",
                "athena:ListNamedQueries",
                "athena:CreateNamedQuery",
                "athena:GetQueryExecution",
                "athena:BatchGetNamedQuery",
                "athena:BatchGetQueryExecution",
                "athena:DeleteWorkGroup",
                "athena:UpdateWorkGroup",
                "athena:GetWorkGroup",
                "athena:CreateWorkGroup"
            ],
            "Resource": [
                "arn:aws:athena:us-east-1:123456789012:workgroup/*"
            ]
        },
        {
            "Effect":"Allow",
            "Action": [
                "athena:CreateDataCatalog",
                "athena:DeleteDataCatalog",
                "athena:GetDataCatalog",
                "athena:GetDatabase",
                "athena:GetTableMetadata",
                "athena:ListDatabases",
                "athena:ListTableMetadata",
                "athena:UpdateDataCatalog"
            ],
        }
    ]
}
```
Example Example policy for management operations on a specified Data Catalog

In the following policy, a user is allowed to create, delete, obtain details, and update a data catalog datacatalogA.

```json
{
"Version": "2012-10-17",
"Statement": [
{
  "Effect": "Allow",
  "Action": [
    "athena:CreateDataCatalog",
    "athena:GetDataCatalog",
    "athena:DeleteDataCatalog",
    "athena:UpdateDataCatalog"
  ],
  "Resource": [
    "arn:aws:athena:us-east-1:123456789012:datacatalog/datacatalogA"
  ]
}
]
}
```

Example Example policy for listing data catalogs

The following policy allows all users to list all data catalogs:

```json
{
"Version": "2012-10-17",
"Statement": [
{
   "Effect": "Allow",
   "Action": [
    "athena:ListDataCatalogs"
   ],
   "Resource": "*
  ]
}
}
```
Example Example policy for metadata operations on data catalogs

The following policy allows metadata operations on data catalogs:

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "athena:GetDatabase",
            "athena:GetTableMetadata",
            "athena:ListDatabases",
            "athena:ListTableMetadata"
         ],
         "Resource": "*"
      }
   ]
}
```

Managing data sources

You can use the Data Sources page of the Athena console to manage the data sources that you create.

To view a data source

2. If the console navigation pane is not visible, choose the expansion menu on the left.
3. In the navigation pane, choose Data sources.
4. From the list of data sources, choose the name of the data source that you want to view.

To edit a data source

1. On the Data sources page, do one of the following:
   - Select the button next to the catalog name, and then choose Actions, Edit.
   - Choose the name of the data source. Then on the details page, choose Actions, Edit.
2. On the Edit page, you can choose a different Lambda function for the data source, change the description, or add custom tags. For more information about tags, see Tagging Athena resources (p. 658).
3. Choose Save.
4. To edit your AwsDataCatalog data source, choose the AwsDataCatalog link to open its details page. Then, on the details page, choose the link to the AWS Glue console where you can edit your catalog.
To share a data source

For information about sharing data sources, visit the following links.

- For non-Hive Lambda-based data sources, see Enabling cross-account federated queries (p. 171).
- For AWS Glue Data Catalogs, see Cross-account access to AWS Glue data catalogs (p. 548).

To delete a data source

1. On the Data sources page, do one of the following:
   - Select the button next to the catalog name, and then choose Actions, Delete.
   - Choose the name of the data source, and then, on the details page, choose Actions, Delete.

   **Note**
   The AwsDataCatalog is the default data source in your account and cannot be deleted.

   You are warned that when you delete a data source, its corresponding data catalog, tables, and views are removed from the query editor. Saved queries that used the data source will no longer run in Athena.

2. To confirm the deletion, type the name of the data source, and then choose Delete.

Connecting to Amazon Athena with ODBC and JDBC drivers

To explore and visualize your data with business intelligence tools, download, install, and configure an ODBC (Open Database Connectivity) or JDBC (Java Database Connectivity) driver.

**Topics**

- Using Athena with the JDBC driver (p. 183)
- Connecting to Amazon Athena with ODBC (p. 185)
- Configuring SSO for ODBC using the Okta plugin and Okta Identity Provider (p. 186)
- Configuring single sign-on using ODBC, SAML 2.0, and the Okta Identity Provider (p. 204)
- Configuring SSO for Azure AD Single-Account Access using ODBC (p. 213)
- Using the Amazon Athena Power BI connector (p. 245)

See also the following AWS Knowledge Center and AWS Big Data Blog topics:

- How can I use my IAM role credentials or switch to another IAM role when connecting to Athena using the JDBC driver?
- Setting up trust between ADFS and AWS and using Active Directory credentials to connect to Amazon Athena with ODBC driver

**Using Athena with the JDBC driver**

You can use a JDBC connection to connect Athena to business intelligence tools and other applications, such as SQL workbench. To do this, use the Amazon S3 links on this page to download, install, and configure the Athena JDBC driver. For information about building the JDBC connection URL, see the
downloadable JDBC driver installation and configuration guide. For permissions information, see Access through JDBC and ODBC connections (p. 536). Starting with version 2.0.24, two versions of the driver are available: one that includes the AWS SDK, and one that does not.

Important
When you use the JDBC driver, be sure to note the following requirements:

- **Open port 444** – Keep port 444, which Athena uses to stream query results, open to outbound traffic. When you use a PrivateLink endpoint to connect to Athena, ensure that the security group attached to the PrivateLink endpoint is open to inbound traffic on port 444. If port 444 is blocked, you may receive the error message [Simba][AthenaJDBC](100123) An error has occurred. Exception during column initialization.

- **athena:GetQueryResultsStream policy** – Add the athena:GetQueryResultsStream policy action to the IAM principals that use the JDBC driver. This policy action is not exposed directly with the API. It is used only with the ODBC and JDBC drivers as part of streaming results support. For an example policy, see AWS managed policy: AWSQuicksightAthenaAccess (p. 534).

- **4.1 drivers** – Driver support for JDBC version 4.1 is discontinued, and further updates will not be released. If you are using a JDBC 4.1 driver, migration to the 4.2 driver is highly recommended. Links to the 4.1 JDBC version will be removed starting in the next calendar year.

### JDBC driver with AWS SDK

The JDBC driver version 2.0.33 complies with the JDBC API 4.1 and 4.2 data standards. For information about checking the version of Java Runtime Environment (JRE) that you use, see the Java documentation. The JRE version depends on the version of the JDBC API that you use with the driver.

Download the driver that matches your version of the JDK and the JDBC data standards:

- The AthenaJDBC41-2.0.33.jar is compatible with JDBC 4.1 and requires JDK 7.0 or later.
- The AthenaJDBC42-2.0.33.jar is compatible with JDBC 4.2 and requires JDK 8.0 or later.

The following .zip file download contains the .jar files for JDBC 4.1 and 4.2 and includes the AWS SDK.

- SimbaAthenaJDBC-2.0.33.1002.zip

### JDBC driver without AWS SDK

The JDBC driver version 2.0.33 complies with the JDBC API 4.1 and 4.2 data standards. For information about checking the version of Java Runtime Environment (JRE) that you use, see the Java documentation. The JRE version depends on the version of the JDBC API that you use with the driver.

Download the driver that matches your version of the JDK and the JDBC data standards. These files do not include the AWS SDK:

- The AthenaJDBC41-2.0.33.jar is compatible with JDBC 4.1 and requires JDK 7.0 or later.
- The AthenaJDBC42-2.0.33.jar is compatible with JDBC 4.2 and requires JDK 8.0 or later.

The following .zip file download contains the .jar files for JDBC 4.1 and 4.2. It does not include the AWS SDK.

- SimbaAthenaJDBC-2.0.33.1003.zip
JDBC driver release notes, license agreement, and notices

After you download the version you need, read the release notes, and review the License Agreement and Notices.

- Release notes
- License agreement
- Notices
- Third-party licenses

JDBC driver documentation

Download the following documentation for the driver:

- JDBC driver installation and configuration guide. Use this guide to install and configure the driver.
- JDBC driver migration guide. Use this guide to migrate from previous versions to the current version.

JDBC driver notes

Using the JDBC driver for multiple data catalogs

To use the JDBC driver for multiple data catalogs with Athena (for example, when using an external Hive metastore (p. 35) or federated queries (p. 60)), include MetadataRetrievalMethod=ProxyAPI in your JDBC connection string.

Connecting to Amazon Athena with ODBC

Use the links on this page to download the Amazon Athena ODBC driver License Agreement, ODBC drivers, and ODBC documentation. For information about the ODBC connection string, see the ODBC Driver Installation and Configuration Guide PDF file, downloadable from this page. For permissions information, see Access through JDBC and ODBC connections (p. 536).

Important
When you use the ODBC driver, be sure to note the following requirements:

- **Open port 444** – Keep port 444, which Athena uses to stream query results, open to outbound traffic. When you use a PrivateLink endpoint to connect to Athena, ensure that the security group attached to the PrivateLink endpoint is open to inbound traffic on port 444.
- **athena:GetQueryResultsStream policy** – Add the athena:GetQueryResultsStream policy action to the IAM principals that use the ODBC driver. This policy action is not exposed directly with the API. It is used only with the ODBC and JDBC drivers as part of streaming results support. For an example policy, see AWS managed policy: AWSQuicksightAthenaAccess (p. 534).

ODBC driver download links

Windows

<table>
<thead>
<tr>
<th>Driver version</th>
<th>Download link</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC 1.1.18.1002 for Windows 32-bit</td>
<td>Windows 32 bit ODBC driver 1.1.18.1002</td>
</tr>
</tbody>
</table>
ODBC driver notes

Connecting Without Using a Proxy

If you want to specify certain hosts that the driver connects to without using a proxy, you can use the optional NonProxyHost property in your ODBC connection string.

The NonProxyHost property specifies a comma-separated list of hosts that the connector can access without going through the proxy server when a proxy connection is enabled, as in the following example:

```
.amazonaws.com,localhost,.example.net,.example.com
```

The NonProxyHost connection parameter is passed to the CURLOPT_NOPROXY curl option. For information about the CURLOPT_NOPROXY format, see CURLOPT_NOPROXY in the curl documentation.

Configuring SSO for ODBC using the Okta plugin and Okta Identity Provider

This page describes how to configure the Amazon Athena ODBC driver and Okta plugin to add single sign-on (SSO) capability using the Okta identity provider.
Prerequisites

Completing the steps in this tutorial requires the following:

- Amazon Athena ODBC driver. For download links, see Connecting to Amazon Athena with ODBC (p. 185).
- An IAM Role that you want to use with SAML. For more information, see Creating a role for SAML 2.0 federation in the IAM User Guide.
- An Okta account. For information, visit Okta.com.

Creating an app integration in Okta

First, use the Okta dashboard to create and configure a SAML 2.0 app for single sign-on to Athena. You can use an existing Redshift application in Okta to configure access to Athena.

To create an app integration in Okta

1. Log in to the Okta admin page at https://admin.okta.com/admin/apps/active
2. In the navigation panel, choose Applications, Applications.
3. On the Applications page, choose Browse App Catalog.
4. On the Browse App Integration Catalog page, in the Use Case section, choose All Integrations.
5. In the search box, enter Amazon Web Services Redshift, and then choose Amazon Web Services Redshift SAML.
6. Choose Add Integration.

7. In the General Settings Required section, for Application label, enter a name for the application. This tutorial uses the name Athena-ODBC-Okta.
8. Choose **Done**.

9. On the page for your Okta application (for example, **Athena-ODBC-Okta**), choose **Sign On**.
10. In the **Settings** section, choose **Edit**.
11. In the Advanced Sign-on Settings section, configure the following values.

- For **IdP ARN and Role ARN**, enter your AWS IDP ARN and Role ARN as comma-separated values. For information about the IAM role format, see Configuring SAML assertions for the authentication response in the IAM User Guide.
- For **Session Duration**, enter a value between 900 and 43200 seconds. This tutorial uses the default of 3600 (1 hour).
### Advanced Sign-on Settings

These fields may be required for a Amazon Web Services Redshift proprietary sign-on option or general setting.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Idp ARN and Role ARN</strong></td>
<td><code>arn:aws:iam::1234567890:saml-provid</code></td>
</tr>
<tr>
<td>Enter your AWS IDP ARN and Role ARN as comma separated values (e.g. &quot;arn:aws:iam::1234567890:saml-provider/OKTA,arn:aws:iam::1234567890:role/SAML_ROLE&quot;).</td>
<td></td>
</tr>
<tr>
<td><strong>Session Duration</strong></td>
<td>3600</td>
</tr>
<tr>
<td>Set the user’s session duration in seconds here. Valid range is 900 to 43200.</td>
<td></td>
</tr>
<tr>
<td><strong>DB User Format (Redshift)</strong></td>
<td><code>${user.username}</code></td>
</tr>
<tr>
<td>EL expression to get DB User value (e.g. &quot;${user.username}&quot;, &quot;${user.firstName}@${user.lastName}@acme.com&quot;)</td>
<td></td>
</tr>
<tr>
<td><strong>Auto Create (Redshift)</strong></td>
<td>![ ]</td>
</tr>
<tr>
<td>AutoCreate Redshift property (Create a new database user if one does not exist)</td>
<td></td>
</tr>
<tr>
<td><strong>Allowed DB Groups (Redshift)</strong></td>
<td></td>
</tr>
<tr>
<td>Comma separated list of allowed user groups. Use <strong>&quot;</strong> to allow all groups, &quot;&quot; to escape comma in group name</td>
<td></td>
</tr>
</tbody>
</table>

The **DbUser Format**, **AutoCreate**, and **Allowed DBGroups** settings aren't used by Athena. You don't have to configure them.

12. Choose **Save**.

### Retrieve ODBC configuration information from Okta

Now that you created the Okta application, you're ready to retrieve the application's ID and IdP host URL. You will require these later when you configure ODBC for connection to Athena.
To retrieve configuration information for ODBC from Okta

1. Choose the **General** tab of your Okta application, and then scroll down to the **App Embed Link** section.

Your **Embed Link** URL is in the following format:

```
https://trial-1234567.okta.com/home/amazon_aws_redshift/Abc1de2fghi3J45kL678/
abc1defghij2klmNo3p4
```

2. From your **Embed Link** URL, extract and save the following pieces:
   - The first segment after `https://`, up to and including `okta.com` (for example, `trial-1234567.okta.com`). This is your IdP host.
   - The last two segments of the URL, including the forward slash in the middle. The segments are two 20-character strings with a mix of numbers and upper and lowercase letters (for example, `Abc1de2fghi3J45kL678/abc1defghij2klmNo3p4`). This is your application ID.

Add a user to the Okta application

Now you're ready to add a user to your Okta application.

To add a user to the Okta application

1. In the left navigation pane, choose **Directory**, and then choose **People**.
2. Choose **Add person**.
3. In the **Add Person** dialog box, enter the following information.

   - Enter values for **First name** and **Last name**. This tutorial uses **test user**.
   - Enter values for **Username** and **Primary email**. This tutorial uses **test@amazon.com** for both. Your security requirements for passwords might vary.
4. Choose **Save**.

Now you're ready to assign the user that you created to your application.

**To assign the user to your application:**

1. In the navigation pane, choose **Applications, Applications**, and then choose the name of your application (for example, **Athena-ODBC-Okta**).
2. Choose **Assign**, and then choose **Assign to People**.
3. Choose the **Assign** option for your user, and then choose **Done**.
4. At the prompt, choose **Save and Go Back**. The dialog box shows the user's status as **Assigned**.

5. Choose **Done**.

6. Choose the **Sign On** tab.

7. Scroll down to the **SAML Signing Certificates** section.

8. Choose **Actions**.

9. Open the context (right-click) menu for **View IdP metadata**, and then choose the browser option to save the file.

10. Save the file with an **.xml** extension.

---

Create an AWS SAML Identity Provider and Role

Now you are ready to upload the metadata XML file to the IAM console in AWS. You will use this file to create an AWS SAML identity provider and role. Use an AWS Services administrator account to perform these steps.

**To create a SAML identity provider and role in AWS**

2. In the navigation pane, choose **Identity providers**, and then choose **Add provider**.
3. On the Add an Identity provider page, for Configure provider, enter the following information.

- For Provider type, choose SAML.
- For Provider name, enter a name for your provider (for example, `AthenaODBCOkta`).
- For Metadata document, use the Choose file option to upload the identity provider (IdP) metadata XML file that you downloaded.
4. Choose Add provider.

**Creating an IAM role for Athena and Amazon S3 access**

Now you are ready to create an IAM role for Athena and Amazon S3 access. You will assign this role to your user. That way, you can provide the user with single sign-on access to Athena.

**To create an IAM role for your user**

1. In the IAM console navigation pane, choose Roles, and then choose Create role.
2. On the Create role page, choose the following options:

- For **Select type of trusted entity**, choose **SAML 2.0 Federation**.
- For **SAML 2.0–based provider**, choose the SAML identity provider that you created (for example, AthenaODBCOkta).
- Select **Allow programmatic and AWS Management Console access**.
3. Choose Next.

4. On the Add Permissions page, for Filter policies, enter AthenaFull, and then press ENTER.

5. Select the AmazonAthenaFullAccess managed policy, and then choose Next.
6. On the Name, review, and create page, for Role name, enter a name for the role (for example, Athena-ODBC-OktaRole), and then choose Create role.

### Configuring the Okta ODBC connection to Athena

Now you're ready to configure the Okta ODBC connection to Athena using the ODBC Data Sources program in Windows.

**To configure your Okta ODBC connection to Athena**

1. In Windows, launch the ODBC Data Sources program.
2. In the ODBC Data Source Administrator program, choose Add.
3. Choose **Simba Athena ODBC Driver**, and then choose **Finish**.

4. In the **Simba Athena ODBC Driver DSN Setup** dialog, enter the values described.
   - For **Data Source Name**, enter a name for your data source (for example, **Athena ODBC 64**).
   - For **Description**, enter a description for your data source.
• For **AWS Region**, enter the AWS Region that you're using (for example, **us-west-1**).
• For **S3 Output Location**, enter the Amazon S3 path where you want your output to be stored.

5. Choose **Authentication Options**.
6. In the **Authentication Options** dialog box, choose or enter the following values.
   • For **Authentication Type**, choose **Okta**.
   • For **User**, enter your Okta user name.
   • For **Password**, enter your Okta password.
   • For **IdP Host**, enter the value that you recorded earlier (for example, **trial-1234567.okta.com**).
   • For **IdP Port**, enter **443**.
- For **App ID**, enter the value that you recorded earlier (the last two segments of your Okta embed link).
- For **Okta App Name**, enter `amazon_aws_redshift`.

![Authentication Options](image-url)
7. Choose **OK**.
8. Choose **Test** to test the connection or **OK** to finish.

### Configuring single sign-on using ODBC, SAML 2.0, and the Okta Identity Provider

To connect to data sources, you can use Amazon Athena with identity providers (IdPs) like PingOne, Okta, OneLogin, and others. Starting with Athena ODBC driver version 1.1.13 and Athena JDBC driver version 2.0.25, a browser SAML plugin is included that you can configure to work with any SAML 2.0 provider. This topic shows you how to configure the Amazon Athena ODBC driver and the browser-based SAML plugin to add single sign-on (SSO) capability using the Okta identity provider.

### Prerequisites

Completing the steps in this tutorial requires the following:

- Athena ODBC driver version 1.1.13 or later. Versions 1.1.13 and later include browser SAML support. For download links, see [Connecting to Amazon Athena with ODBC](#).
- An IAM Role that you want to use with SAML. For more information, see [Creating a role for SAML 2.0 federation](#) in the IAM User Guide.
- An Okta account. For information, visit okta.com.

### Creating an app integration in Okta

First, use the Okta dashboard to create and configure a SAML 2.0 app for single sign-on to Athena.

**To use the Okta dashboard to set up single sign-on for Athena**

1. Login to the Okta admin page at https://admin.okta.com/admin/apps/active.
2. In the navigation pane, choose Applications, Applications.
3. On the Applications page, choose Create App Integration.

   ![Applications](#)

4. In the Create a new app integration dialog box, for Sign-in method, select SAML 2.0, and then choose Next.
5. On the **Create SAML Integration** page, in the **General Settings** section, enter a name for the application. This tutorial uses the name **Athena SSO**.

6. Choose **Next**.

7. On the **Configure SAML** page, in the **SAML Settings** section, enter the following values:
   - For **Single sign on URL**, enter `http://localhost:7890/athena`
   - For **Audience URI**, enter `urn:amazon:webservices`
8. For **Attribute Statements (optional)**, enter the following two name/value pairs. These are required mapping attributes.

- For **Name**, enter the following URL:

  https://aws.amazon.com/SAML/Attributes/Role

  For **Value**, enter the name of your IAM role. For information about the IAM role format, see Configuring SAML assertions for the authentication response in the IAM User Guide.

- For **Name**, enter the following URL:
https://aws.amazon.com/SAML/Attributes/RoleSessionName

For **Value**, enter `user.email`.

### Attribute Statements (optional)

<table>
<thead>
<tr>
<th>Name</th>
<th>Name format</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="https://aws">https://aws</a>.</td>
<td>Unspecified</td>
<td>YOUR_ROLE</td>
</tr>
<tr>
<td><a href="https://aws">https://aws</a>.</td>
<td>Unspecified</td>
<td>user.email</td>
</tr>
</tbody>
</table>

9. Choose **Next**, and then choose **Finish**.

When Okta creates the application, it also creates your login URL, which you will retrieve next.

### Getting the login URL from the Okta dashboard

Now that your application has been created, you can obtain its login URL and other metadata from the Okta dashboard.

**To get the login URL from the Okta dashboard**

1. In the Okta navigation pane, choose **Applications, Applications**.
2. Choose the application for which you want to find the login URL (for example, **AthenaSSO**).
3. On the page for your application, choose **Sign On**.

4. Choose **View Setup Instructions**.
5. On the How to Configure SAML 2.0 for Athena SSO page, find the URL for Identity Provider Issuer. Some places in the Okta dashboard refer to this URL as the SAML issuer ID.

6. Copy or store the value for Identity Provider Single Sign-On URL.

In the next section, when you configure the ODBC connection, you will provide this value as the Login URL connection parameter for the browser SAML plugin.
Configuring the browser SAML ODBC connection to Athena

Now you are ready to configure the browser SAML connection to Athena using the ODBC Data Sources program in Windows.

To configure the browser SAML ODBC connection to Athena

1. In Windows, launch the ODBC Data Sources program.
2. In the ODBC Data Source Administrator program, choose Add.

3. Choose Simba Athena ODBC Driver, and then choose Finish.
4. In the **Simba Athena ODBC Driver DSN Setup** dialog, enter the values described.
1. For **Data Source Name**, enter a name for your data source (for example, **Athena ODBC 64**).
2. For **Description**, enter a description for your data source.
3. For **AWS Region**, enter the AWS Region that you are using (for example, **us-west-1**).
4. For **S3 Output Location**, enter the Amazon S3 path where you want your output to be stored.
5. Choose **Authentication Options**.
6. In the **Authentication Options** dialog box, choose or enter the following values.
Configuring SSO using ODBC, SAML 2.0, and Okta

Authentication Options

Authentication Type: BrowserSAML

User: 

Password: 

Password Options...

Session Token: 

Preferred Role: 

Session Duration: 

Login URL: http://www.okta.com/

Listen Port: 7890

Timeout (sec): 60

Use HTTP Proxy For IdP Host

SSL Insecure

OK Cancel
• For **Authentication Type**, choose **BrowserSAML**.
• For **Login URL**, enter the **Identity Provider Single Sign-On URL** that you obtained from the Okta dashboard.
• For **Listen Port**, enter 7890.
• For **Timeout (sec)**, enter a connection timeout value in seconds.

7. Choose **OK** to close **Authentication Options**.
8. Choose **Test** to test the connection, or **OK** to finish.

### Configuring SSO for Azure AD Single-Account Access using ODBC

AWS single-account access enables you to federate Azure AD to a single AWS account and use Azure AD to manage access to AWS IAM roles. For each AWS account, Azure AD administrators federate to AWS IAM, assign users or groups to the account, and configure Azure AD to authorize role access.

**Note**
The directions provided here are specific to Athena. For general instructions on configuring Azure Active Directory SSO for use with AWS, see Tutorial: Azure Active Directory single sign-on (SSO) integration with AWS Single-Account Access in the Microsoft documentation.

**Important**
This tutorial shows you how to configure Azure Active Directory and ODBC for single sign-on to Athena with and without MFA support. The MFA and non-MFA settings are slightly different. For the **Reply URL** and **SAML:aud** values, non-MFA support uses defaults, but MFA support uses `http://localhost/athena`.

### Step 1: Creating an Azure Enterprise Application

To begin, you create an enterprise application in the Azure AD portal. When configuration is complete, your application will control your Azure users' access to Athena.

**To create an AWS single-account access application in the Azure AD portal**

2. Choose **Azure Active Directory**.
3. In the navigation pane, choose Enterprise applications.
4. Choose **New application**.
5. In the **Browse Azure AD Gallery** page search box, enter **AWS**, and then choose **AWS Single-Account Access**.
6. In the right pane, for **Name**, enter a name for your application. The default name is **AWS Single-Account Access**.

7. Choose **Create**.
The creation process can take a minute or two to complete.

9. For **Select a single sign-on method**, choose **SAML**.

10. At the **Save single sign-on setting** prompt, choose **Yes**.

11. For the **Basic SAML Configuration** section, do one of the following:

   - If your application does not require multifactor authentication, for **Identifier (Entity ID)** and **Reply URL**, confirm that the URL `https://signin.aws.amazon.com/saml` is prepopulated, as the following image shows:
• If your application requires MFA, choose Edit, and then, in the Basic SAML Configuration pane, perform the following tasks:

1. For Identifier (Entity ID), enter `urn:amazon:webservices`
2. For Reply URL, enter `http://localhost/athena`
3. At the top of the Basic SAML Configuration pane, choose Save, and then choose the X icon to close the pane.
4. At the Test single sign-on prompt, choose No, I'll test later.

12. Notice that, in the User Attributes & Claims section, the necessary claims are already configured for you.

13. In the SAML Signing Certificate section, for Federation Metadata XML, choose Download, and then save the XML file in a secure location.
Later, you upload this metadata file to the IAM console to enable communication between Athena and Azure.

Adding the Delegated User.Read Permission to your Application

Next, you give delegated read permission to your enterprise application to sign in with SSO and read user profile information.

To add delegated read permissions to your enterprise application

1. At the top of the Azure portal, choose Default Directory.
2. In the navigation pane, choose App registrations.
3. Choose the **All applications** tab, and then choose your enterprise application.
4. In the navigation pane, choose **API permissions**.
5. In the **Configured permissions** section, choose **Add a permission**.
6. In the **Request API permissions** pane on the right, choose **Microsoft Graph**.
7. For **What type of permissions does your application require?**, choose **Delegated permissions**.
8. In the **Selection permissions** search box, enter **User.Read**.
9. Expand the **User** section.
10. Select **User.Read**.
11. Choose **Add permissions**.
12. In the **Configured permissions** section, choose **Grant admin consent for Default Directory**, and then choose **Yes** at the confirmation prompt.

On the **API Permissions** page, the **Configured permissions** section shows that the **User.Read** permission has been delegated to your application.

13. For more information about the **User.Read** permission, choose the **User.Read** link in the **Configured permissions** section.

**For MFA: Specifying the Athena Redirect URI**

Perform the steps in this section only if your connection will use multifactor authentication.
To specify a redirect URI in Azure AD for multifactor authentication

1. At the top of the Azure portal, choose Default Directory.
2. In the navigation pane, choose App registrations.
3. Choose the All applications tab, and then choose your enterprise application.
4. In the left navigation pane, choose Authentication.
5. In the Platform configurations section, choose Add a platform.

Platform configurations

Depending on the platform or device this application is targeted to, redirect URLs, specific authentication settings, or fields specific to the platform configuration are required. To configure single sign-on, including SAML-based sign-on, try the following:

#### Web

Redirect URIs

The URLs we will accept as destinations when returning users after authenticating or signing out users. Also referred to as redirect URIs.

- ![restrictions](https://signin.aws.amazon.com/saml)
- https://signin.aws.amazon.com/saml

Add URI

Front-channel logout URL

This is where we send a request to have the application clear the session and log the user out.

6. In the Configure platforms pane on the right, in the Mobile and desktop applications section, choose Mobile and desktop applications.
7. In the Configure Desktop + devices pane, for Custom redirect URIs, enter http://localhost/athena, and then choose Configure.
You have now set up the enterprise application in Azure. You are ready to set up the IAM SAML identity provider and IAM role to use this Azure federation.

Step 2: Using the IAM Console to Configure Azure Access to Athena

In this section, you perform the following tasks in AWS in the IAM management console:

- Create an identity provider for AWS using the metadata that you downloaded from Azure AD.
- Create an IAM role. When you create the role, you attach policies that allow access to Athena and Amazon S3. You will assign this role to your Azure AD users so that you can provide them with single sign-on access to Athena.
- Enable Azure AD to list AWS IAM Roles by 1) creating a policy that grants list roles permissions and then 2) by creating an IAM user for Azure AD. When you create the IAM user for Azure AD, you attach the list roles policy.

Creating an Identity Provider

In this section, you use the metadata from the enterprise application that you created in Azure to create a SAML identity provider in the AWS IAM console.
To create a SAML identity provider in IAM using your Azure metadata

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.

2. In the navigation pane, choose Identity providers, and then choose Add provider.

3. On the Add an Identity provider page, for Configure provider, enter the following information.
• For **Provider type**, choose **SAML**.
• For **Provider name**, enter a name for your provider (for example, **AthenaAzureADSSO**).
• For **Metadata document**, use the **Choose file** option to upload the identity provider (IdP) metadata XML file that you downloaded from Azure AD.

4. Choose **Add provider**.

**Creating an IAM Role for Athena and Amazon S3 Access**

Next, you create an IAM role for Athena and Amazon S3 access. You will assign this role to your Azure AD users so that you can provide them with single sign-on access to Athena.

**To create an IAM role for Athena and Amazon S3 access**

1. In the IAM console navigation pane, choose **Roles**, and then choose **Create role**.
2. Choose SAML 2.0 federation.

3. In the Choose a SAML 2.0 provider section, for SAML provider, choose the SAML identity provider that you created (for example, AthenaAzureADSSO.)

4. Select Allow programmatic and AWS Management Console access.

5. For the SAML: aud Attribute, do one of the following:
   
   - If your application does not require multifactor authentication, make sure that Value is set to https://signin.aws.amazon.com/saml (the default).
   - If your application requires MFA, for Value, enter http://localhost/athena.

6. Choose Next: Permissions.

7. In the Attach permissions policies section, attach the necessary policies for your application. This tutorial attaches the AmazonAthenaFullAccess and AmazonS3FullAccess policies.
8. Choose Next: Tags.
9. Choose Next: Review.
10. For Role name, enter a name for your role (for example, AthenaAzureADRole).
11. Confirm the policies that you chose are present, and then choose Create role.

Enabling Azure to List Roles

To secure federated access, the Azure AD service must know the roles that are available in your account. To enable this functionality, you create a policy that grants list roles permissions and then create an IAM user for the Azure AD service. When you create the IAM user for the Azure AD service, you attach the list roles policy to the user.

Create a List Roles Policy

In this step, you use the IAM console to create a policy that grants list roles permissions.
To create a list roles policy

1. In the navigation pane of the IAM console, choose **Policies**, and then choose **Create policy**.
2. On the **Create policy** page, choose **JSON**.
3. Paste the following ListRoles policy into the JSON editor.

   ```json
   {
     "Version": "2012-10-17",
     "Statement": [
       {
         "Effect": "Allow",
         "Action": [
           "iam:ListRoles"
         ],
         "Resource": "*"
       }
     ]
   }
   
```

4. Choose **Next: Tags**.
5. Choose **Next: Review**.
6. On the **Review policy** page, enter a name for the policy. This walkthrough uses the name `AthenaListRolesAzure`. 

230
Choose Create policy.

Create a User for the Azure AD service and attach the ListRoles policy to it

Now you are ready to create an IAM user that Azure can use to list the roles that are available to it.

To create a user for the Azure AD service and attach the list roles policy to the user

1. In the navigation pane of the IAM console, choose Users, and then choose Add users.
2. On the Add user page, for User name, enter a user name. This tutorial uses the name AthenaAzureADRoleProvision.
3. In the Select AWS Access type section, choose Access key - Programmatic access.
4. Choose Next: Permissions.
5. On the Set permissions page, choose Attach existing policies directly.
6. For **Filter policies**, enter the name of the list roles policy that you just created (for example, *AthenaListRolesAzure*), and then select the check box for the policy.

7. Choose **Next: Tags**.

8. Choose **Next: Review**.

9. Choose **Create user**.

10. On the **Success** page, copy the **Access key ID** and **Secret access key**. You will add these to Azure next.

### Step 3: Configuring Role Provisioning in Azure AD

Next, in the Azure AD portal, you add AWS access keys to Azure and configure the Azure provisioning service so that it can import roles from AWS automatically.

**To configure role provisioning in the Azure AD portal**

2. In the navigation pane, choose **Enterprise Applications**.
3. Choose your enterprise application.
4. In the navigation pane, choose **Provisioning**.
5. Choose **Get started**.
6. In the **Provisioning** pane, for **Provisioning Mode**, choose **Automatic**.
7. Expand the **Admin Credentials** section, and then enter the required information.
• For `clientsecret`, enter the access key ID of the IAM user for Azure that you created in the IAM console (in this walkthrough, the access key ID for `AthenaAzureADRoleProvision`.)

• For `Secret Token`, enter the corresponding secret access key.

8. Choose Test Connection and verify that test completes successfully.
9. At the top of the Provisioning pane, choose Save.

10. Choose the X icon to close the Provisioning pane where you entered the admin credentials and tested the connection. You can return to this page later by choosing Edit provisioning from the main Provisioning page.

11. On the main Provisioning page for your application, choose Start provisioning. The provisioning process can take a few minutes. To update the status, you can choose Refresh at the top of the Provisioning page.

When the provisioning is done, the Azure AD portal reports the completion under Current cycle status and displays the number of roles that it imported.
Step 4: Creating and Configuring Users in Azure AD

At this point, you have created your AWS SSO application. And, thanks to the provisioning that you performed, Azure AD is aware of the IAM roles available to it. This means that when you create users in Azure AD, you can now assign them to your AWS SSO application and to the IAM role that provides access to Athena and Amazon S3.

In this section, you:

- Create the test user **Athena Test User** in Azure AD.
- Assign **Athena Test User** to your **AWS Single-Account Access** application.
- Assign **Athena Test User** to the **AthenaAzureADRole** so that it has access to Athena.

Creating a Test User in Azure AD

The following procedure creates an Athena test user called **Athena Test User** in the Azure AD portal.

To create a test user in Azure AD

1. At the top of the Azure portal, choose Default Directory.
2. In the Default Directory navigation pane, choose Users.
3. Choose **New user** to create new Azure AD test user.

4. Choose **Create user** or **Invite user**.

5. Enter the information for the new user, and then choose **Create** or **Invite**. For this tutorial, a user called **Athena Test User** was created using the **Invite** option.
Assigning Athena Users to Your Azure AD Application and IAM Role

Now that you have an Athena test user, you can assign it to your SSO application and to the AthenaAzureADRole that allows the user to access Athena and Amazon S3.

**To assign an Azure AD user to your application and IAM role**

1. In the Azure AD portal, choose Default Directory, Enterprise applications, and then choose your application.
2. In the navigation pane, choose Users and Groups, and then choose Add user/group.

3. On the Add Assignment page, for Users, choose None Selected.
4. In the Users pane on the right, find and select your Athena test user, and then choose Select.

5. On the Add Assignment page, for Select a role, choose None Selected.

6. In the Select a role pane on the right, find and select your AthenaAzureADRole. Both the role and the identity provider name are displayed.
7. Choose **Select**.

8. On the **Add Assignment** page, choose **Assign**.

The **Athena Test User** appears in the list of users for the **AWS Single-Account Access** application with the **AthenaAzureADRole** assigned.
Step 5: Configuring ODBC

To configure your ODBC connection, you obtain your client and tenant IDs from the Azure portal, and then enter these values using the ODBC Data Sources program in Microsoft Windows.

Azure AD Portal: Obtaining Your Client and Tenant ID

You can obtain your client and tenant IDs from the App registrations section of the Azure portal. The following procedure shows how.

To get your client and tenant ID from the Azure AD portal

2. Choose the All applications tab, and then choose your enterprise application.
3. In the Essentials section, find your UUIDs for Application (client) ID and Directory (tenant) ID.
4. Use the **Copy to clipboard** icon to copy and store these values. You will use these values in the next section to configure the Athena ODBC connection to Azure AD.

**Windows: Configuring Athena ODBC for Azure AD**

This section shows you how to use the ODBC Data Sources program in Windows to configure your Athena ODBC connection for Azure AD.

**To configure a browser Azure AD ODBC connection to Athena**

1. Install the ODBC driver for Athena. For download links, see *Connecting to Amazon Athena with ODBC* (p. 185).
2. In Windows, launch the **ODBC Data Sources** program.
3. In the **ODBC Data Source Administrator** program, choose **Add**.

4. Choose **Simba Athena ODBC Driver**, and then choose **Finish**.
5. In the **Simba Athena ODBC Driver DSN Setup** dialog, enter the values described.
• For **Data Source Name**, enter a name for your data source (for example, **Athena ODBC 64**).
• For **Description**, enter a description for your data source.
• For **AWS Region**, enter the AWS Region that you are using (for example, **us-west-1**).
• For **S3 Output Location**, enter the Amazon S3 path where you want your output to be stored.

6. Choose **Authentication Options**.
7. In the **Authentication Options** dialog box, choose or enter the values described.
For **Authentication Type**, choose one of the following:
- If you are using MFA, choose **BrowserAzureAD**.
- If you are not using MFA, choose either **BrowserAzureAD** or **AzureAD**.

For **User**, enter the Azure AD user name.

For **Password**, enter the Azure AD password.

For **Tenant ID**, enter the value for the **Directory (tenant) ID** that you copied from the **Overview** page of your enterprise application in the Azure AD portal.

For **Client ID**, enter the value for the **Application (client) ID** that you copied from the **Overview** page of your enterprise application in the Azure AD portal.
8. Choose **OK**.
9. Choose **Test** to test the connection, or **OK** to finish.

**ODBC.INI File Configuration Example**

After you finish, the ODBC.INI file in your Windows directory (for example, `C:\Windows\ODBC.INI`) should have a DSN configuration entry like the following:

```ini
[Athena BrowserAzureAD Test]
Description=Simba Athena ODBC Driver (64-bit) DSN
Driver=/opt/simba/athenaodbc/lib/64/libathenaodbc_sb64.so
AwsRegion=[YOUR-REGION]
Workgroup=[YOUR-WORKGROUP]
S3OutputLocation=[YOUR-OUTPUTLOCATION]
AuthenticationType=BrowserAzureAD
UID=[AzureAD USERNAME]
PWD=[AzureAD PASSWORD]
idp_tenant=[YOUR TENANT ID]
client_id=[YOUR CLIENT ID]
```

**Using the Amazon Athena Power BI connector**

On Windows operating systems, you can use the Microsoft Power BI connector for Amazon Athena to analyze data from Amazon Athena in Microsoft Power BI Desktop. For information about Power BI, see Microsoft power BI. After you publish content to the Power BI service, you can use the July 2021 or later release of **Power BI gateway** to keep the content up to date through on-demand or scheduled refreshes.

**Prerequisites**

Before you begin, make sure that your environment meets the following requirements. The Amazon Athena ODBC driver is required.

- AWS account
- Permissions to use Athena (p. 536)
- Amazon Athena ODBC driver (p. 185)
- Power BI desktop

**Capabilities supported**

- **Import** – Selected tables and columns are imported into Power BI Desktop for querying.
- **DirectQuery** – No data is imported or copied into Power BI Desktop. Power BI Desktop queries the underlying data source directly.
- **Power BI gateway** – An on-premises data gateway in your AWS account that works like a bridge between the Microsoft Power BI Service and Athena. The gateway is required to see your data on the Microsoft Power BI Service.

**Connect to Amazon Athena**

To connect Power BI desktop to your Amazon Athena data, perform the following steps.

**To connect to Athena data from power BI desktop**

1. Launch Power BI Desktop.
2. Do one of the following:
   - Choose **File, Get Data**
   - From the **Home** ribbon, choose **Get Data**.

3. In the search box, enter **Athena**.

4. Select **Amazon Athena**, and then choose **Connect**.

![Get Data](image)

5. On the **Amazon Athena** connection page, enter the following information.
   - For **DSN**, enter the name of the ODBC DSN that you want to use. For instructions on configuring your DSN, see the **ODBC driver documentation** (p. 186).
   - For **Data Connectivity mode**, choose a mode that is appropriate for your use case, following these general guidelines:
     - For smaller datasets, choose **Import**. When using Import mode, Power BI works with Athena to import the contents of the entire dataset for use in your visualizations.
     - For larger datasets, choose **DirectQuery**. In DirectQuery mode, no data is downloaded to your workstation. While you create or interact with a visualization, Microsoft Power BI works with Athena to dynamically query the underlying data source so that you're always viewing current data. For more information about DirectQuery, see [Use DirectQuery in power BI desktop](#) in the Microsoft documentation.
6. Choose **OK**.

7. At the prompt to configure data source authentication, choose either **Use Data Source Configuration** or **AAD Authentication**, and then choose **Connect**.

Your data catalog, databases, and tables appear in the **Navigator** dialog box.
8. In the **Display Options** pane, select the check box for the dataset that you want to use.

9. If you want to transform the dataset before you import it, go to the bottom of the dialog box and choose **Transform Data**. This opens the Power Query Editor so that you can filter and refine the set of data you want to use.

10. Choose **Load**. After the load is complete, you can create visualizations like the one in the following image. If you selected **DirectQuery** as the import mode, Power BI issues a query to Athena for the visualization that you requested.
Setting up an on-premises gateway

You can publish dashboards and datasets to the Power BI service so that other users can interact with them through web, mobile, and embedded apps. To see your data in the Microsoft Power BI Service, you install the Microsoft Power BI on-premises data gateway in your AWS account. The gateway works like a bridge between the Microsoft Power BI Service and Athena.
To download, install, and test an on-premises data gateway

1. Visit the Microsoft power BI gateway download page and choose either personal mode or standard mode. Personal mode is useful for testing the Athena connector locally. Standard mode is appropriate in a multiuser production setting.

2. To install an on-premises gateway (either personal or standard mode), see Install an on-premises data gateway in the Microsoft documentation.

3. To test the gateway, follow the steps in Use custom data connectors with the on-premises data gateway in the Microsoft documentation.

For more information about on-premises data gateways, see the following Microsoft resources.

- What is an on-premises data gateway?
- Guidance for deploying a data gateway for power BI

For an example of configuring Power BI Gateway for use with Athena, see the AWS Big Data Blog article Creating dashboards quickly on Microsoft power BI using amazon Athena.
Creating databases and tables

Amazon Athena supports a subset of data definition language (DDL) statements and ANSI SQL functions and operators to define and query external tables where data resides in Amazon Simple Storage Service.

When you create a database and table in Athena, you describe the schema and the location of the data, making the data in the table ready for real-time querying.

To improve query performance and reduce costs, we recommend that you partition your data and use open source columnar formats for storage in Amazon S3, such as Apache parquet or ORC.

Topics
• Creating databases in Athena (p. 251)
• Creating tables in Athena (p. 251)
• Names for tables, databases, and columns (p. 254)
• Reserved keywords (p. 256)
• Table location in Amazon S3 (p. 257)
• Columnar storage formats (p. 259)
• Converting to columnar formats (p. 259)
• Partitioning data in Athena (p. 260)
• Partition projection with Amazon Athena (p. 265)

Creating databases in Athena

A database in Athena is a logical grouping for tables you create in it. Creating a database in the Athena console query editor is straightforward.

To create a database using the Athena query editor

2. On the Editor tab, enter the Hive data definition language (DDL) command `CREATE DATABASE myDatabase`. Replace `myDatabase` with the name that you want to use. For restrictions on database names, see Names for tables, databases, and columns (p. 254).
3. Choose Run or press Ctrl+ENTER.
4. To make your database the current database, select it from the Database menu on the left of the query editor.

For information about controlling permissions to Athena databases, see Fine-grained access to databases and tables in the AWS Glue Data Catalog (p. 540).

Creating tables in Athena

You can run DDL statements in the Athena console, using a JDBC or an ODBC driver, or using the Athena Create table form (p. 253).

When you create a new table schema in Athena, Athena stores the schema in a data catalog and uses it when you run queries.
Athena uses an approach known as *schema-on-read*, which means a schema is projected on to your data at the time you run a query. This eliminates the need for data loading or transformation.

Athena does not modify your data in Amazon S3.

Athena uses Apache Hive to define tables and create databases, which are essentially a logical namespace of tables.

When you create a database and table in Athena, you are simply describing the schema and the location where the table data are located in Amazon S3 for read-time querying. Database and table, therefore, have a slightly different meaning than they do for traditional relational database systems because the data isn’t stored along with the schema definition for the database and table.

When you query, you query the table using standard SQL and the data is read at that time. You can find guidance for how to create databases and tables using Apache Hive documentation, but the following provides guidance specifically for Athena.

The maximum query string length is 256 KB.

Hive supports multiple data formats through the use of serializer-deserializer (SerDe) libraries. You can also define complex schemas using regular expressions. For a list of supported SerDe libraries, see Supported SerDes and data formats (p. 303).

# Considerations and limitations

Following are some important limitations and considerations for tables in Athena.

## Requirements for tables in Athena and data in Amazon S3

When you create a table, you specify an Amazon S3 bucket location for the underlying data using the LOCATION clause. Consider the following:

- Athena can only query the latest version of data on a versioned Amazon S3 bucket, and cannot query previous versions of the data.
- You must have the appropriate permissions to work with data in the Amazon S3 location. For more information, see Access to Amazon S3 (p. 536).
- Athena supports querying objects that are stored with multiple storage classes in the same bucket specified by the LOCATION clause. For example, you can query data in objects that are stored in different Storage classes (Standard, Standard-IA and Intelligent-Tiering) in Amazon S3.
- Athena supports Requester Pays buckets. For information how to enable Requester Pays for buckets with source data you intend to query in Athena, see Create a workgroup (p. 640).
- Athena does not support querying the data in the S3 Glacier flexible retrieval or S3 Glacier Deep Archive storage classes. Objects in the S3 Glacier Flexible Retrieval and S3 Glacier Deep Archive storage classes are ignored. As an alternative, you can use the Amazon S3 Glacier Instant Retrieval storage class, which is queryable by Athena. For more information, see Amazon S3 Glacier instant retrieval storage class.

For information about storage classes, see Storage classes, Changing the storage class of an object in Amazon S3, Transitioning to the GLACIER storage class (object archival), and Requester Pays buckets in the Amazon Simple Storage Service User Guide.

- If you issue queries against Amazon S3 buckets with a large number of objects and the data is not partitioned, such queries may affect the Get request rate limits in Amazon S3 and lead to Amazon S3 exceptions. To prevent errors, partition your data. Additionally, consider tuning your Amazon S3 request rates. For more information, see Request rate and performance considerations.
- If you use the AWS Glue CreateTable API operation or the AWS CloudFormation AWS::Glue::Table template to create a table for use in Athena without specifying the TableType property and then run
a DDL query like `SHOW CREATE TABLE` or `MSCK REPAIR TABLE`, you can receive the error message 
`FAILED: NullPointerException Name is null.`

To resolve the error, specify a value for the `TableInput TableType` attribute as part of the AWS Glue 
CreateTable API call or AWS CloudFormation template. Possible values for `TableType` include 
`EXTERNAL_TABLE` or `VIRTUAL_VIEW`.

This requirement applies only when you create a table using the AWS Glue CreateTable API 
operation or the AWS::Glue::Table template. If you create a table for Athena by using a DDL 
statement or an AWS Glue crawler, the `TableType` property is defined for you automatically.

**Functions supported**

The functions supported in Athena queries correspond to those in Trino and Presto. For information 
about individual functions, see the functions and operators section in the Trino or Presto 
documentation.

**Transactional data transformations are not supported**

Athena does not support transaction-based operations (such as the ones found in Hive or Presto) on 
table data. For a full list of keywords not supported, see Unsupported DDL (p. 731).

**Operations that change table states are ACID**

When you create, update, or delete tables, those operations are guaranteed ACID-compliant. For 
example, if multiple users or clients attempt to create or alter an existing table at the same time, only 
one will be successful.

**Tables are EXTERNAL**

Except when creating governed tables (p. 500) or Iceberg (p. 504) tables, always use the EXTERNAL 
keyword. If you use `CREATE TABLE` without the EXTERNAL keyword for non-governed, non-Iceberg 
tables, Athena issues an error. When you drop a table in Athena, only the table metadata is removed; the 
data remains in Amazon S3.

**Creating tables using AWS Glue or the Athena console**

You can create tables in Athena by using AWS Glue, the add table form, or by running a DDL statement in 
the Athena query editor.

**To create a table using the AWS Glue crawler**

2. In the query editor, next to Tables and views, choose Create, and then choose AWS Glue crawler.
3. Follow the steps on the Add crawler page of the AWS Glue console to add a crawler.

   For more information, see Using AWS Glue crawlers (p. 26).

**To create a table using the Athena create table form**

2. In the query editor, next to Tables and views, choose Create, and then choose S3 bucket data.
Amazon Athena User Guide
Showing table information

3. In the Create Table From S3 bucket data form, enter the information to create your table, and then
choose Create table. For more information about the ﬁelds in the form, see Adding a table using a
form (p. 22).

To create a table using Hive DDL
1. From the Database menu, choose the database for which you want to create a table. If you don't
specify a database in your CREATE TABLE statement, the table is created in the database that is
currently selected in the query editor.
2. Enter a statement like the following in the query editor, and then choose Run, or press Ctrl+ENTER.
CREATE EXTERNAL TABLE IF NOT EXISTS cloudfront_logs (
`Date` Date,
Time STRING,
Location STRING,
Bytes INT,
RequestIP STRING,
Method STRING,
Host STRING,
Uri STRING,
Status INT,
Referrer STRING,
OS String,
Browser String,
BrowserVersion String
) ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
"input.regex" = "^(?!#)([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s
+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s+[^\(]+[\(]([^\;]+).*\%20([^\/]+)[\/](.*)$"
) LOCATION 's3://athena-examples-MyRegion/cloudfront/plaintext/';

Showing table information
After you have created a table in Athena, its name displays in the Tables list on the left. To show
information about the table and manage it, choose the vertical three dots next to the table name in the
Athena console.
• Preview table – Shows the ﬁrst 10 rows of all columns by running the SELECT * FROM
"database_name"."table_name" LIMIT 10 statement in the Athena query editor.
• Generate table DDL – Generates a DDL statement that you can use to re-create the table by running
the SHOW CREATE TABLE (p. 759) table_name statement in the Athena query editor.
• Load partitions – Runs the MSCK REPAIR TABLE (p. 756) table_name statement in the Athena
query editor. This option is available only if the table has partitions.
• Insert into editor – Inserts the name of the table into the query editor at the current editing location.
• Delete table – Displays a conﬁrmation dialog box asking if you want to delete the table. If you agree,
runs the DROP TABLE (p. 755) table_name statement in the Athena query editor.
• Table properties – Shows the table name, database name, time created, and whether the table has
encrypted data.

Names for tables, databases, and columns
Use these tips for naming items in Athena.

254


Use lower case for table names and table column names in Athena

Athena accepts mixed case in DDL and DML queries, but lower cases the names when it executes the query. For this reason, avoid using mixed case for table or column names, and do not rely on casing alone in Athena to distinguish such names. For example, if you use a DDL statement to create a column named Castle, the column created will be lowercased to castle. If you then specify the column name in a DML query as Castle or CASTLE, Athena will lowercase the name for you to run the query, but display the column heading using the casing that you chose in the query.

Database, table, and column names must be less than or equal to 255 characters long.

Special characters

Special characters other than underscore (_) are not supported. For more information, see the Apache Hive LanguageManual DDL documentation.

Important

Although you may succeed in creating table, view, database, or column names that contain special characters other than underscore by enclosing them in backtick (`) characters, subsequent DDL or DML queries that reference them can fail.

Names that begin with an underscore

When creating tables, use backticks to enclose table, view, or column names that begin with an underscore. For example:

```
CREATE EXTERNAL TABLE IF NOT EXISTS `_myunderscoretable`
(`_id` string, `_index` string)
LOCATION 's3://my-athena-data/
```

Table, view, or column names that begin with numbers

When running SELECT, CTAS, or VIEW queries, put quotation marks around identifiers like table, view, or column names that start with a digit. For example:

```
CREATE OR REPLACE VIEW "123view" AS
SELECT "123columnone", "123columntwo"
FROM "234table"
```

Column names and complex types

For complex types, only alphanumeric characters, underscore (_), and period (.) are allowed in column names. To create a table and mappings for keys that have restricted characters, you can use a custom DDL statement. For more information, see the article Create tables in Amazon Athena from nested JSON and mappings using JSONSerDe in the AWS Big Data Blog.

Reserved words

Certain reserved words in Athena must be escaped. To escape reserved keywords in DDL statements, enclose them in backticks (`). To escape reserved keywords in SQL SELECT statements and in queries on views (p. 361), enclose them in double quotes (").
Reserved keywords

When you run queries in Athena that include reserved keywords, you must escape them by enclosing them in special characters. Use the lists in this topic to check which keywords are reserved in Athena.

To escape reserved keywords in DDL statements, enclose them in backticks (`). To escape reserved keywords in SQL SELECT statements and in queries on views (p. 361), enclose them in double quotes (").

- List of reserved keywords in DDL statements (p. 256)
- List of reserved keywords in SQL SELECT statements (p. 256)
- Examples of queries with reserved words (p. 257)

List of reserved keywords in DDL statements

Athena uses the following list of reserved keywords in its DDL statements. If you use them without escaping them, Athena issues an error. To escape them, enclose them in backticks (`).

You cannot use DDL reserved keywords as identifier names in DDL statements without enclosing them in backticks (`).

<table>
<thead>
<tr>
<th>Reserved Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL, ALTER, AND, ARRAY, AS, AUTHORIZATION, BETWEEN, BIGINT, BINARY, BOOLEAN, BOTH, BY, CASE, CASE WHEN, CAST, CHAR, COLUMN, CONF, CONSTRAINT, COMMIT, CREATE, CROSS, CUBE, CURRENT, CURRENT_DATE, CURRENT_TIMESTAMP, CURSOR, DATABASE, DATE, DAYOFWEEK, DECIMAL, DELETE, DESCRIBE, DISTINCT, DOUBLE, DROP, ELSE, END, EXCEPT, EXECUTE, EXISTS, EXTENDED, EXTRACT, FALSE, FETCH, FLOAT, FLOOR, FOLLOWING, FOR, FOREIGN, FROM, FULL, FUNCTION, GRANT, GROUP, GROUPING, HAVING, IF, IMPORT, IN, INNER, INSERT, INT, INTEGER, INTERVAL, INTO, IS, JOIN, LATERAL, LEFT, LESS, LIKE, LOCAL, MACRO, MAP, MORE, NONE, NOT, NULL, NUMERIC, OF, ON, ONLY, OR, ORDER, OUTER, OVER, PARTIALSCAN, PARTITION, PERCENT, PRECEDING, PRECISION, PRESERVE, PRIMARY, PROCEDURE, RANGE, READS, REDUCE, REGEXP, REFERENCES, REVOKE, RIGHT, RLIKE, ROLLBACK, ROLLUP, ROW, ROWS, SELECT, SET, SMALLINT, START, TABLE, TABLESAMPLE, THEN, TIME, TIMESTAMP, TO, TRANSFORM, TRIGGER, TRUE, TRUNCATE, UNBOUNDED, UNION, UNIQUEJOIN, UPDATE, USER, USING, UTC_TIMESTAMP, VALUES, VARCHAR, VIEWS, WHEN, WHERE, WINDOW, WITH</td>
</tr>
</tbody>
</table>

List of reserved keywords in SQL SELECT statements

Athena uses the following list of reserved keywords in SQL SELECT statements and in queries on views.

If you use these keywords as identifiers, you must enclose them in double quotes (") in your query statements.

<table>
<thead>
<tr>
<th>Reserved Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER, AND, AS, BETWEEN, BY, CASE, CAST, CONSTRAINT, CREATE, CROSS, CUBE, CURRENT_DATE, CURRENT_PATH, CURRENT_TIME, CURRENT_TIMESTAMP, CURRENT_USER, DEALLOCATE, DELETE, DESCRIBE, DISTINCT, DROP, ELSE, END, ESCAPE, EXCEPT, EXECUTE, EXISTS, EXTRACT, FALSE, FIRST, FOR, FROM, FULL, GROUP, GROUPING, HAVING, IN, INNER, INSERT, INTERSECT, INTO, IS, JOIN, LAST, LEFT, LIKE, LOCALTIME, LOCALTIMESTAMP, NATURAL, NORMALIZE, NOT, NULL, OF, ON, OR, ORDER, OUTER, PREPARE, RECURSIVE, RIGHT, ROLLUP, SELECT, TABLE, THEN, TRUE,</td>
</tr>
</tbody>
</table>
Examples of queries with reserved words

The query in the following example uses backticks (`) to escape the DDL-related reserved keywords `partition` and `date` that are used for a table name and one of the column names:

```sql
CREATE EXTERNAL TABLE `partition` (
  `date` INT,
  col2 STRING
) PARTITIONED BY (year STRING)
STORED AS TEXTFILE
LOCATION 's3://test_bucket/test_examples/';
```

The following example queries include a column name containing the DDL-related reserved keywords in ALTER TABLE ADD PARTITION and ALTER TABLE DROP PARTITION statements. The DDL reserved keywords are enclosed in backticks (`):

```sql
ALTER TABLE test_table
ADD PARTITION (`date` = '2018-05-14')

ALTER TABLE test_table
DROP PARTITION (`partition` = 'test_partition_value')
```

The following example query includes a reserved keyword (end) as an identifier in a SELECT statement. The keyword is escaped in double quotes:

```sql
SELECT *
FROM TestTable
WHERE "end" != nil;
```

The following example query includes a reserved keyword (first) in a SELECT statement. The keyword is escaped in double quotes:

```sql
SELECT "itemId"."first"
FROM testTable
LIMIT 10;
```

Table location in Amazon S3

When you run a CREATE TABLE query in Athena, you register your table with the AWS Glue Data Catalog. (If you are using Athena's older internal catalog, we highly recommend that you upgrade (p. 31) to the AWS Glue Data Catalog.)

To specify the path to your data in Amazon S3, use the LOCATION property, as shown in the following example:

```sql
CREATE EXTERNAL TABLE `test_table`(
  ...
) ROW FORMAT ... STORED AS INPUTFORMAT ... OUTPUTFORMAT ...
```
LOCATION s3://bucketname/folder/

- For information about naming buckets, see Bucket restrictions and limitations in the Amazon Simple Storage Service User Guide.
- For information about using folders in Amazon S3, see Using folders in the Amazon Simple Storage Service User Guide.

The LOCATION in Amazon S3 specifies *all* of the files representing your table.

**Important**

Athena reads *all* data stored in the Amazon S3 folder that you specify. If you have data that you do *not* want Athena to read, do not store that data in the same Amazon S3 folder as the data that you do want Athena to read. If you are leveraging partitioning, to ensure Athena scans data within a partition, your WHERE filter must include the partition. For more information, see Table location and partitions (p. 258).

When you specify the LOCATION in the CREATE TABLE statement, use the following guidelines:

- Use a trailing slash.
- You can use a path to an Amazon S3 folder or an Amazon S3 access point alias. For information about Amazon S3 access point aliases, see Using a bucket-style alias for your access point in the Amazon S3 User Guide.

**Use:**

s3://bucketname/folder/

s3://access-point-name-metadata-s3alias/folder/

Do not use any of the following items for specifying the LOCATION for your data.

- Do not use filenames, underscores, wildcards, or glob patterns for specifying file locations.
- Do not add the full HTTP notation, such as s3.amazonaws.com to the Amazon S3 bucket path.
- Do not use empty folders like // in the path, as follows: S3://bucketname/folder//folder/. While this is a valid Amazon S3 path, Athena does not allow it and changes it to s3://bucketname/folder/folder/, removing the extra /.

**Do not use:**

s3://path_to_bucket
s3://path_to_bucket/*
s3://path_to_bucket/mySpecialFile.dat
s3://bucketname/prefix/filename.csv
s3://test-bucket.s3.amazonaws.com
s3://bucket/prefix/
arn:aws:s3:::bucketname/prefix
s3://arn:aws:s3:<account_id>:accesspoint/<accesspointname>
https://<accesspointname>-<number>.s3-accesspoint.<region>.amazonaws.com

Table location and partitions

Your source data may be grouped into Amazon S3 folders called *partitions* based on a set of columns. For example, these columns may represent the year, month, and day the particular record was created.
When you create a table, you can choose to make it partitioned. When Athena runs a SQL query against a non-partitioned table, it uses the LOCATION property from the table definition as the base path to list and then scan all available files. However, before a partitioned table can be queried, you must update the AWS Glue Data Catalog with partition information. This information represents the schema of files within the particular partition and the LOCATION of files in Amazon S3 for the partition.

- To learn how the AWS Glue crawler adds partitions, see How does a crawler determine when to create partitions? in the AWS Glue Developer Guide.
- To learn how to configure the crawler so that it creates tables for data in existing partitions, see Using multiple data sources with crawlers (p. 27).
- You can also create partitions in a table directly in Athena. For more information, see Partitioning data in Athena (p. 260).

When Athena runs a query on a partitioned table, it checks to see if any partitioned columns are used in the WHERE clause of the query. If partitioned columns are used, Athena requests the AWS Glue Data Catalog to return the partition specification matching the specified partition columns. The partition specification includes the LOCATION property that tells Athena which Amazon S3 prefix to use when reading data. In this case, only data stored in this prefix is scanned. If you do not use partitioned columns in the WHERE clause, Athena scans all the files that belong to the table's partitions.

For examples of using partitioning with Athena to improve query performance and reduce query costs, see Top performance tuning tips for Amazon Athena.

Columnar storage formats

Apache Parquet and ORC are columnar storage formats that are optimized for fast retrieval of data and used in AWS analytical applications.

Columnar storage formats have the following characteristics that make them suitable for using with Athena:

- **Compression by column**, with compression algorithm selected for the column data type to save storage space in Amazon S3 and reduce disk space and I/O during query processing.
- **Predicate pushdown** in Parquet and ORC enables Athena queries to fetch only the blocks it needs, improving query performance. When an Athena query obtains specific column values from your data, it uses statistics from data block predicates, such as max/min values, to determine whether to read or skip the block.
- **Splitting of data** in Parquet and ORC allows Athena to split the reading of data to multiple readers and increase parallelism during its query processing.

To convert your existing raw data from other storage formats to Parquet or ORC, you can run CREATE TABLE AS SELECT (CTAS) (p. 281) queries in Athena and specify a data storage format as Parquet or ORC, or use the AWS Glue Crawler.

Converting to columnar formats

Your Amazon Athena query performance improves if you convert your data into open source columnar formats, such as Apache parquet or ORC.

Options for easily converting source data such as JSON or CSV into a columnar format include using CREATE TABLE AS (p. 281) queries or running jobs in AWS Glue.
• You can use CREATE TABLE AS (CTAS) queries to convert data into Parquet or ORC in one step. For an example, see Example: Writing query results to a different format on the Examples of CTAS queries (p. 286) page.

• For information about running an AWS Glue job to transform CSV data to Parquet, see the section "Transform the data from CSV to Parquet format" in the AWS Big Data blog post Build a data lake foundation with AWS Glue and Amazon S3. AWS Glue supports using the same technique to convert CSV data to ORC, or JSON data to either Parquet or ORC.

Partitioning data in Athena

By partitioning your data, you can restrict the amount of data scanned by each query, thus improving performance and reducing cost. You can partition your data by any key. A common practice is to partition the data based on time, often leading to a multi-level partitioning scheme. For example, a customer who has data coming in every hour might decide to partition by year, month, date, and hour. Another customer, who has data coming from many different sources but that is loaded only once per day, might partition by a data source identifier and date.

Athena can use Apache Hive style partitions, whose data paths contain key value pairs connected by equal signs (for example, country=us/... or year=2021/month=01/day=26/...). Thus, the paths include both the names of the partition keys and the values that each path represents. To load new Hive partitions into a partitioned table, you can use the MSCK REPAIR TABLE (p. 756) command, which works only with Hive-style partitions.

Athena can also use non-Hive style partitioning schemes. For example, CloudTrail logs and Kinesis Data Firehose delivery streams use separate path components for date parts such as data/2021/01/26/us/6fc7845e.json. For such non-Hive style partitions, you use ALTER TABLE ADD PARTITION (p. 733) to add the partitions manually.

Considerations and limitations

When using partitioning, keep in mind the following points:

• If you query a partitioned table and specify the partition in the WHERE clause, Athena scans the data only from that partition. For more information, see Table location and partitions (p. 258).

• If the partition name is within the WHERE clause of the subquery, Athena currently does not filter the partition and instead scans all data from the partitioned table.

• If you issue queries against Amazon S3 buckets with a large number of objects and the data is not partitioned, such queries may affect the GET request rate limits in Amazon S3 and lead to Amazon S3 exceptions. To prevent errors, partition your data. Additionally, consider tuning your Amazon S3 request rates. For more information, see Best practices design patterns: Optimizing Amazon S3 performance.

• Partition locations to be used with Athena must use the s3 protocol (for example, s3://DOC-EXAMPLE-BUCKET/folder/). In Athena, locations that use other protocols (for example, s3a://DOC-EXAMPLE-BUCKET/folder/) will result in query failures when MSCK REPAIR TABLE queries are run on the containing tables.

• Make sure that the Amazon S3 path is in lower case instead of camel case (for example, userId instead of user Id). If the S3 path is in camel case, MSCK REPAIR TABLE doesn't add the partitions to the AWS Glue Data Catalog. For more information, see MSCK REPAIR TABLE (p. 756).

• Because MSCK REPAIR TABLE scans both a folder and its subfolders to find a matching partition scheme, be sure to keep data for separate tables in separate folder hierarchies. For example, suppose you have data for table A in s3://table-a-data and data for table B in s3://table-a-data/table-b-data. If both tables are partitioned by string, MSCK REPAIR TABLE will add the partitions
for table B to table A. To avoid this, use separate folder structures like s3://table-a-data and s3://table-b-data instead. Note that this behavior is consistent with Amazon EMR and Apache Hive.

- If you are using the AWS Glue Data Catalog with Athena, see AWS Glue endpoints and quotas for service quotas on partitions per account and per table.
  - Although Athena supports querying AWS Glue tables that have 10 million partitions, Athena cannot read more than 1 million partitions in a single scan. In such scenarios, partition indexing can be beneficial. For more information, see the AWS Big Data Blog article Improve Amazon Athena query performance using AWS Glue Data Catalog partition indexes.
- To request a partitions quota increase if you are using the AWS Glue Data Catalog, visit the Service Quotas console for AWS Glue.

Creating and loading a table with partitioned data

To create a table that uses partitions, use the `PARTITIONED BY` clause in your `CREATE TABLE` statement. The `PARTITIONED BY` clause defines the keys on which to partition data, as in the following example. The `LOCATION` clause specifies the root location of the partitioned data.

```
CREATE EXTERNAL TABLE users (first string,
last string,
username string)
PARTITIONED BY (id string)
STORED AS parquet
LOCATION 's3://DOC-EXAMPLE-BUCKET/folder/
```

After you create the table, you load the data in the partitions for querying. For Hive style partitions, you run `MSCK REPAIR TABLE` (p. 756). For non-Hive style partitions, you use `ALTER TABLE ADD PARTITION` (p. 733) to add the partitions manually.

Preparing Hive style and non-Hive style data for querying

The following sections show how to prepare Hive style and non-Hive style data for querying in Athena.

**Scenario 1: Data stored on Amazon S3 in Hive format**

In this scenario, partitions are stored in separate folders in Amazon S3. For example, here is the partial listing for sample ad impressions output by the `aws s3 ls` command, which lists the S3 objects under a specified prefix:

```
aws s3 ls s3://elasticmapreduce/samples/hive-ads/tables/impressions/
```

PRE dt=2009-04-12-13-00/
PRE dt=2009-04-12-13-05/
PRE dt=2009-04-12-13-10/
PRE dt=2009-04-12-13-15/
PRE dt=2009-04-12-13-20/
PRE dt=2009-04-12-14-00/
PRE dt=2009-04-12-14-05/
PRE dt=2009-04-12-14-10/
PRE dt=2009-04-12-14-15/
PRE dt=2009-04-12-14-20/
PRE dt=2009-04-12-15-00/
PRE dt=2009-04-12-15-05/

Here, logs are stored with the column name (dt) set equal to date, hour, and minute increments. When you give a DDL with the location of the parent folder, the schema, and the name of the partitioned column, Athena can query data in those subfolders.

Create the table

To make a table from this data, create a partition along 'dt' as in the following Athena DDL statement:

```
CREATE EXTERNAL TABLE impressions (
    requestBeginTime string,
    adId string,
    impressionId string,
    referrer string,
    userAgent string,
    userCookie string,
    ip string,
    number string,
    processId string,
    browserCookie string,
    requestEndTime string,
    timers struct<modelLookup:string, requestTime:string>,
    threadId string,
    hostname string,
    sessionId string)
PARTITIONED BY (dt string)
ROW FORMAT  serde 'org.apache.hive.hcatalog.data.JsonSerDe'
LOCATION 's3://elasticmapreduce/samples/hive-ads/tables/impressions/';
```

This table uses Hive's native JSON serializer-deserializer to read JSON data stored in Amazon S3. For more information about the formats supported, see Supported SerDes and data formats (p. 303).

Run MSCK REPAIR TABLE

After you run the CREATE TABLE query, run the MSCK REPAIR TABLE command in the Athena query editor to load the partitions, as in the following example.

```
MSCK REPAIR TABLE impressions
```

After you run this command, the data is ready for querying.

Query the data

Query the data from the impressions table using the partition column. Here's an example:

```
SELECT dt,impressionId FROM impressions WHERE dt<'2009-04-12-14-00' and dt>='2009-04-12-13-00' ORDER BY dt DESC LIMIT 100
```

This query should show results similar to the following:

<table>
<thead>
<tr>
<th>dt</th>
<th>impressionId</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009-04-12-13-20</td>
<td>ap3HcVKAWfXtgIPu6WpuUFafL0DQEc</td>
</tr>
<tr>
<td>2009-04-12-13-20</td>
<td>17uchtodoS9kdeQP1x0XTk151uRsV</td>
</tr>
<tr>
<td>2009-04-12-13-20</td>
<td>J0UF15CtRwv16W8sVcghqE5h0nkgtP</td>
</tr>
<tr>
<td>2009-04-12-13-20</td>
<td>NQ2XP0J0dvBvCX30pb4Xvq35A4qxxH</td>
</tr>
</tbody>
</table>
Scenario 2: Data is not partitioned in Hive format

In the following example, the `aws s3 ls` command shows ELB (p. 451) logs stored in Amazon S3. Note how the data layout does not use `key=value` pairs and therefore is not in Hive format. (The --recursive option for the `aws s3 ls` command specifies that all files or objects under the specified directory or prefix be listed.)

```
aws s3 ls s3://athena-examples-myregion/elb/plaintext/ --recursive
```

```
2016-11-23 17:54:46  11789573 elb/plaintext/2015/01/01/part-r-00000-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:46    8776899 elb/plaintext/2015/01/01/part-r-00001-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:46    9309800 elb/plaintext/2015/01/01/part-r-00002-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:47    9412570 elb/plaintext/2015/01/01/part-r-00003-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:47   10725938 elb/plaintext/2015/01/01/part-r-00004-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:47    9439710 elb/plaintext/2015/01/01/part-r-00005-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:47          0 elb/plaintext/2015/01/01_$folder$
2016-11-23 17:54:48   11979218 elb/plaintext/2015/01/02/part-r-00006-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:48   11211291 elb/plaintext/2015/01/02/part-r-00007-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:48    9673393 elb/plaintext/2015/01/02/part-r-00008-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:48   11979218 elb/plaintext/2015/01/02/part-r-00009-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:48    9546833 elb/plaintext/2015/01/02/part-r-00010-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:48   11360522 elb/plaintext/2015/01/02/part-r-00011-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:48   10731284 elb/plaintext/2015/01/02/part-r-00012-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:48    9984735 elb/plaintext/2015/01/02/part-r-00013-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:48    9290089 elb/plaintext/2015/01/02/part-r-00014-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:48    9336462 elb/plaintext/2015/01/02/part-r-00015-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:48    9178313 elb/plaintext/2015/01/02/part-r-00016-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:48    9895582 elb/plaintext/2015/01/02/part-r-00017-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:48          0 elb/plaintext/2015/01/02_$folder$
2016-11-23 17:54:49    8633768 elb/plaintext/2015/01/03/part-r-00018-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:49    9181626 elb/plaintext/2015/01/03/part-r-00019-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:49    9173813 elb/plaintext/2015/01/03/part-r-00020-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:49    9895582 elb/plaintext/2015/01/03/part-r-00021-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
```

263
Run ALTER TABLE ADD PARTITION

Because the data is not in Hive format, you cannot use the MSCK REPAIR TABLE command to add the partitions to the table after you create it. Instead, you can use the ALTER TABLE ADD PARTITION (p. 733) command to add each partition manually. For example, to load the data in \texttt{s3://athena-examples-myregion/elb/plaintext/2015/01/01/}, you can run the following query. Note that a separate partition column for each Amazon S3 folder is not required, and that the partition key value can be different from the Amazon S3 key.

\begin{verbatim}
ALTER TABLE elb_logs_raw_native_part ADD PARTITION (dt='2015-01-01') location 's3://athena-examples-us-west-1/elb/plaintext/2015/01/01/';
\end{verbatim}

If a partition already exists, you receive the error Partition already exists. To avoid this error, you can use the IF NOT EXISTS clause. For more information, see ALTER TABLE ADD PARTITION (p. 733). To remove a partition, you can use ALTER TABLE DROP PARTITION (p. 735).
Partition projection

To avoid having to manage partitions, you can use partition projection. Partition projection is an option for highly partitioned tables whose structure is known in advance. In partition projection, partition values and locations are calculated from table properties that you configure rather than read from a metadata repository. Because the in-memory calculations are faster than remote look-up, the use of partition projection can significantly reduce query runtimes.

For more information, see Partition projection with Amazon Athena (p. 265).

Additional resources

- For information about partitioning options for Kinesis Data Firehose data, see Amazon Kinesis Data Firehose example (p. 277).
- You can automate adding partitions by using the JDBC driver (p. 183).
- You can use CTAS and INSERT INTO to partition a dataset. For more information, see Using CTAS and INSERT INTO for ETL and data analysis (p. 290).

Partition projection with Amazon Athena

You can use partition projection in Athena to speed up query processing of highly partitioned tables and automate partition management.

In partition projection, partition values and locations are calculated from configuration rather than read from a repository like the AWS Glue Data Catalog. Because in-memory operations are often faster than remote operations, partition projection can reduce the runtime of queries against highly partitioned tables. Depending on the specific characteristics of the query and underlying data, partition projection can significantly reduce query runtime for queries that are constrained on partition metadata retrieval.

Pruning and projection for heavily partitioned tables

Partition pruning gathers metadata and “prunes” it to only the partitions that apply to your query. This often speeds up queries. Athena uses partition pruning for all tables with partition columns, including those tables configured for partition projection.

Normally, when processing queries, Athena makes a GetPartitions call to the AWS Glue Data Catalog before performing partition pruning. If a table has a large number of partitions, using GetPartitions can affect performance negatively. To avoid this, you can use partition projection. Partition projection allows Athena to avoid calling GetPartitions because the partition projection configuration gives Athena all of the necessary information to build the partitions itself.

Using partition projection

To use partition projection, you specify the ranges of partition values and projection types for each partition column in the table properties in the AWS Glue Data Catalog or in your external Hive metastore (p. 35). These custom properties on the table allow Athena to know what partition patterns to expect when it runs a query on the table. During query execution, Athena uses this information to project the partition values instead of retrieving them from the AWS Glue Data Catalog or external Hive metastore. This not only reduces query execution time but also automates partition management because it removes the need to manually create partitions in Athena, AWS Glue, or your external Hive metastore.

Important

Enabling partition projection on a table causes Athena to ignore any partition metadata registered to the table in the AWS Glue Data Catalog or Hive metastore.
Use cases

Scenarios in which partition projection is useful include the following:

- Queries against a highly partitioned table do not complete as quickly as you would like.
- You regularly add partitions to tables as new date or time partitions are created in your data. With partition projection, you configure relative date ranges that can be used as new data arrives.
- You have highly partitioned data in Amazon S3. The data is impractical to model in your AWS Glue Data Catalog or Hive metastore, and your queries read only small parts of it.

Projectable partition structures

Partition projection is most easily configured when your partitions follow a predictable pattern such as, but not limited to, the following:

- **Integers** – Any continuous sequence of integers such as $[1, 2, 3, 4, \ldots, 1000]$ or $[0500, 0550, 0600, \ldots, 2500]$.
- **Dates** – Any continuous sequence of dates or datetimes such as $[20200101, 20200102, \ldots, 20201231]$ or $[1-1-2020 00:00:00, 1-1-2020 01:00:00, \ldots, 12-31-2020 23:00:00]$.
- **Enumerated values** – A finite set of enumerated values such as airport codes or AWS Regions.
- **AWS service logs** – AWS service logs typically have a known structure whose partition scheme you can specify in AWS Glue and that Athena can therefore use for partition projection.

Customizing the partition path template

By default, Athena builds partition locations using the form `s3://<bucket>/<table-root>/partition-col-1=<partition-col-1-val>/partition-col-2=<partition-col-2-val>/`, but if your data is organized differently, Athena offers a mechanism for customizing this path template. For steps, see *Specifying custom S3 storage locations* (p. 271).

Considerations and limitations

The following considerations apply:

- Partition projection eliminates the need to specify partitions manually in AWS Glue or an external Hive metastore.
- When you enable partition projection on a table, Athena ignores any partition metadata in the AWS Glue Data Catalog or external Hive metastore for that table.
- If a projected partition does not exist in Amazon S3, Athena will still project the partition. Athena does not throw an error, but no data is returned. However, if too many of your partitions are empty, performance can be slower compared to traditional AWS Glue partitions. If more than half of your projected partitions are empty, it is recommended that you use traditional partitions.
- Queries for values that are beyond the range bounds defined for partition projection do not return an error. Instead, the query runs, but returns zero rows. For example, if you have time-related data that starts in 2020 and is defined as `projection.timestamp.range='2020/01/01,NOW'`, a query like `SELECT * FROM table-name WHERE timestamp = '2019/02/02'` will complete successfully, but return zero rows.
- Partition projection is usable only when the table is queried through Athena. If the same table is read through another service such as Amazon Redshift Spectrum or Amazon EMR, the standard partition metadata is used.
• Because partition projection is a DML-only feature, SHOW PARTITIONS does not list partitions that are projected by Athena but not registered in the AWS Glue catalog or external Hive metastore.

• Athena does not use the table properties of views as configuration for partition projection. To work around this limitation, configure and enable partition projection in the table properties for the tables that the views reference.

• Lake Formation data filters cannot be used with partition projection in Athena.

Video

The following video shows how to use partition projection to improve the performance of your queries in Athena.

Partition projection with Amazon Athena

Topics

• Setting up partition projection (p. 267)
• Supported types for partition projection (p. 271)
• Dynamic ID partitioning (p. 275)
• Amazon Kinesis Data Firehose example (p. 277)

Setting up partition projection

Setting up partition projection in a table's properties is a two-step process:

1. Specify the data ranges and relevant patterns for each partition column, or use a custom template.
2. Enable partition projection for the table.

This section shows how to set these table properties for AWS Glue. To set them, you can use the AWS Glue console, Athena CREATE TABLE (p. 740) queries, or AWS Glue API operations. The following procedure shows how to set the properties in the AWS Glue console.

To configure and enable partition projection using the AWS Glue console

1. Sign in to the AWS Management Console and open the AWS Glue console at https://console.aws.amazon.com/glue/.
2. Choose the Tables tab.

On the Tables tab, you can edit existing tables, or choose Add tables to create new ones. For information about adding tables manually or with a crawler, see Working with tables on the AWS Glue console in the AWS Glue Developer Guide.

3. In the list of tables, choose the link for the table that you want to edit.
4. Choose **Edit table**.

5. In the **Edit table details** dialog box, in the **Table properties** section, for each partitioned column, add the following key-value pair:
   a. For **Key**, add `projection.columnName.type`.
   b. For **Value**, add one of the supported types: `enum`, `integer`, `date`, or `injected`. For more information, see **Supported types for partition projection** (p. 271).

6. Following the guidance in **Supported types for partition projection** (p. 271), add additional key-value pairs according to your configuration requirements.

The following example table configuration configures the `year` column for partition projection, restricting the values that can be returned to a range from 2010 through 2016.
7. Add a key-value pair to enable partition projection. For **Key**, enter `projection.enabled`, and for its **Value**, enter `true`.

![Table properties]

**Note**
You can disable partition projection on this table at any time by setting `projection.enabled` to `false`.

8. When you are finished, choose **Apply**.

9. In the Athena Query Editor, test query the columns that you configured for the table.

The following example query uses `SELECT DISTINCT` to return the unique values from the `year` column. The database contains data from 1987 to 2016, but the `projection.year.range` property restricts the values returned to the years 2010 to 2016.
Note
If you set `projection.enabled` to `true` but fail to configure one or more partition columns, you receive an error message like the following:
HIVE_METASTORE_ERROR: Table `database_name.table_name` is configured for partition projection, but the following partition columns
Specifying custom S3 storage locations

When you edit table properties in AWS Glue, you can also specify a custom Amazon S3 path template for the projected partitions. A custom template enables Athena to properly map partition values to custom Amazon S3 file locations that do not follow a typical .../column=value/... pattern.

Using a custom template is optional. However, if you use a custom template, the template must contain a placeholder for each partition column. Templated locations must end with a forward slash so that the partitioned data files live in a “folder” per partition.

To specify a custom partition location template

1. Following the steps to configure and enable partition projection using the AWS Glue console (p. 267), add an additional a key-value pair that specifies a custom template as follows:
   a. For Key, enter storage.location.template.
   b. For Value, specify a location that includes a placeholder for every partition column. Make sure that each placeholder (and the S3 path itself) is terminated by a single forward slash.

   The following example template values assume a table with partition columns a, b, and c.

   ```
   s3://bucket/table_root/a=${a}/${b}/some_static_subdirectory/${c}/
   s3://bucket/table_root/c=${c}/${b}/some_static_subdirectory/${a}/${b}/${c}/${c}/
   ```

   For the same table, the following example template value is invalid because it contains no placeholder for column c.

   ```
   s3://bucket/table_root/a=${a}/${b}/some_static_subdirectory/
   ```

2. Choose Apply.

Supported types for partition projection

A table can have any combination of enum, integer, date, or injected partition column types.

**Enum type**

Use the enum type for partition columns whose values are members of an enumerated set (for example, airport codes or AWS Regions).

Define the partition properties in the table as follows:

<table>
<thead>
<tr>
<th>Property name</th>
<th>Example values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>projection.columnName.type</td>
<td>enum</td>
<td>Required. The projection type to use for column columnName. The value must be enum (case</td>
</tr>
</tbody>
</table>
## Supported types for partition projection

<table>
<thead>
<tr>
<th>Property name</th>
<th>Example values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>insensitive) to signal the use of the enum type. Leading and trailing white space is allowed.</td>
<td>projection.columnName.values</td>
<td>Required. A comma-separated list of enumerated partition values for column columnName. Any white space is considered part of an enum value.</td>
</tr>
</tbody>
</table>

### Note

As a best practice we recommend limiting the use of enum based partition projections to a few dozen or less. Although there is no specific limit for enum projections, the total size of your table's metadata cannot exceed the AWS Glue limit of about 1MB when gzip compressed. Note that this limit is shared across key parts of your table like column names, location, storage format, and others. If you find yourself using more than a few dozen unique IDs in your enum projection, consider an alternative approach such as bucketing into a smaller number of unique values in a surrogate field. By trading off cardinality, you can control the number of unique values in your enum field.

### Integer type

Use the integer type for partition columns whose possible values are interpretable as integers within a defined range. Projected integer columns are currently limited to the range of a Java signed long (-2^63 to 2^63-1 inclusive).

<table>
<thead>
<tr>
<th>Property name</th>
<th>Example values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>projection.columnName.type integer</td>
<td></td>
<td>Required. The projection type to use for column columnName. The value must be integer (case insensitive) to signal the use of the integer type. Leading and trailing white space is allowed.</td>
</tr>
<tr>
<td>projection.columnName.range</td>
<td>-1,8675309</td>
<td>Required. A two-element comma-separated list that provides the minimum and maximum range values to be returned by queries on the column columnName. These values are inclusive, can be negative, and can have leading zeroes. Leading and trailing white space is allowed.</td>
</tr>
<tr>
<td>projection.columnName.interval</td>
<td>5</td>
<td>Optional. A positive integer that specifies the interval between successive partition values for the column columnName. For example, a range value of &quot;1,3&quot; with an interval value of &quot;1&quot; produces the values 1, 2, and 3.</td>
</tr>
</tbody>
</table>
## Supported types for partition projection

<table>
<thead>
<tr>
<th>Property name</th>
<th>Example values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The same range value with an interval value of &quot;2&quot; produces the values 1 and 3, skipping 2. Leading and trailing white space is allowed. The default is 1.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| projection.columnName.digits | 5 | Optional. A positive integer that specifies the number of digits to include in the partition value's final representation for column columnName. For example, a range value of "1,3" that has a digits value of "1" produces the values 1, 2, and 3. The same range value with a digits value of "2" produces the values 01, 02, and 03. Leading and trailing white space is allowed. The default is no static number of digits and no leading zeroes. |

### Date type

Use the date type for partition columns whose values are interpretable as dates (with optional times) within a defined range.

**Important**
Projected date columns are generated in Coordinated Universal Time (UTC) at query execution time.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Example values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>projection.columnName.type</td>
<td>date</td>
<td>Required. The projection type to use for column columnName. The value must be date (case insensitive) to signal the use of the date type. Leading and trailing white space is allowed.</td>
</tr>
</tbody>
</table>

| projection.columnName.range | 201701, 201812 | Required. A two-element, comma-separated list which provides the minimum and maximum range values for the column columnName. These values are inclusive and can use any format compatible with the Java java.time.* date types. Both the minimum and maximum values must use the same format. The format specified in the .format property must be the format used for these values. |

This column can also contain relative date strings, formatted in this regular expression pattern:

```
\s*NOW\s*(\+[\-])\s*([0-9]+)\s*(YEARS?|MONTHS?|WEEKS?|DAYS?|HOURS?|MINUTES?|SECONDS?)\s*)?|
```
<table>
<thead>
<tr>
<th>Property name</th>
<th>Example values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Description</td>
</tr>
<tr>
<td></td>
<td>White spaces are allowed, but in date literals are considered part of the date strings themselves.</td>
</tr>
<tr>
<td>projection.columnName.format</td>
<td>yyyyMMdd</td>
</tr>
<tr>
<td>projection.columnName.format</td>
<td>dd-MM-yyyy</td>
</tr>
<tr>
<td>projection.columnName.format</td>
<td>dd-MM-yyyy-HH-mm-ss</td>
</tr>
<tr>
<td>projection.columnName.interval</td>
<td>1</td>
</tr>
<tr>
<td>projection.columnName.interval</td>
<td>5</td>
</tr>
<tr>
<td>projection.columnName.interval.unit</td>
<td>YEARS</td>
</tr>
<tr>
<td>projection.columnName.interval.unit</td>
<td>MONTHS</td>
</tr>
<tr>
<td>projection.columnName.interval.unit</td>
<td>WEEKS</td>
</tr>
<tr>
<td>projection.columnName.interval.unit</td>
<td>DAYS</td>
</tr>
<tr>
<td>projection.columnName.interval.unit</td>
<td>HOURS</td>
</tr>
<tr>
<td>projection.columnName.interval.unit</td>
<td>MINUTES</td>
</tr>
<tr>
<td>projection.columnName.interval.unit</td>
<td>SECONDS</td>
</tr>
<tr>
<td>projection.columnName.interval.unit</td>
<td>MILLISECONDS</td>
</tr>
</tbody>
</table>

**Injected type**

Use the injected type for partition columns with possible values that cannot be procedurally generated within some logical range but that are provided in a query's WHERE clause as a single value.

It is important to keep in mind the following points:

- Queries on injected columns fail if a filter expression is not provided for each injected column.
- Queries with multiple values for a filter expression on an injected column succeed only if the values are disjunct.
- Only columns of string type are supported.
<table>
<thead>
<tr>
<th>Property name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>projection.columnName.type</code></td>
<td><code>injected</code></td>
<td>Required. The projection type to use for the column <code>columnName</code>. Only the string type is supported. The value specified must be injected (case insensitive). Leading and trailing white space is allowed.</td>
</tr>
</tbody>
</table>

For more information, see [Injection](p. 275).

## Dynamic ID partitioning

You might have tables partitioned on a unique identifier column that has the following characteristics:

- Adds new values frequently, perhaps automatically.
- Cannot be easily generated. They might be user names or device IDs of varying composition or length, not sequential integers within a defined range.

For such partitioning schemes, the enum projection type would be impractical for the following reasons:

- You would have to modify the table properties each time a value is added to the table.
- A single table property would have millions of characters or more.
- Projection requires that all partition columns be configured for projection. This requirement could not be avoided for only one column.

To overcome these limitations, you can use injection or bucketing.

### Injection

If your query pattern on a dynamic ID dataset always specifies a single value for the high cardinality partition column, you can use value injection. Injection avoids the need to project the full partition space.

Imagine that you want to partition an IoT dataset on a UUID field that has extremely high cardinality like `device_id`. The field has the following characteristics:

- An extremely high number (potentially billions) of values.
- Because its values are random strings, it is not projectable using other projection methods.
- The extremely large number of partitions cannot be stored in commonly used metastores.

However, if all of your queries include a WHERE clause that filters for only a single `device_id`, you can use the following approach in your CREATE TABLE statement.

```sql
...
PARTITIONED BY
(
    device_id STRING
)
LOCATION "s3://bucket/prefix/"
TBLPROPERTIES
(
    "projection.enabled" = "true",
    "projection.device_id.type" = "injected",
    "storage.location.template" = "s3://bucket/prefix/${device_id}"
)
```
A SELECT query on a table like this looks like the following:

```sql
SELECT
col1,
col2,...,
device_id
FROM
table
WHERE
device_id = "b6319dc2-48c1-4cd5-a0a3-a1969f7b48f7"
AND (    
col1 > 0
    or col2 < 10
)
```

In the example, Athena projects only a single partition for any given query. This avoids the need to store and act upon millions or billions of virtual partitions only to find one partition and read from it.

The following example shows a query that uses multiple injected values for a column:

```sql
SELECT * FROM logs
WHERE service IN ('kafka' OR 'SQS' OR 'email' OR ...)
AND datehour >= '2020/01/01/00'
```

**Bucketing**

In the bucketing technique, you use a fixed set of bucket values rather than the entire set of identifiers for your partitioning. If you can map an identifier to a bucket, you can use this mapping in your queries. You still benefit as when you partition on the identifiers themselves.

Bucketing has the following advantages over injection:

- You can specify more than one value at a time for a field in the WHERE clause.
- You can continue to use your partitions with more traditional metastores.

Using the scenario in the previous example and assuming 1 million buckets, identified by an integer, the CREATE TABLE statement becomes the following.

```sql
... PARTITIONED BY
(
    BUCKET_ID BIGINT
)
LOCATION "s3://bucket/prefix/"
TBLPROPERTIES
(
    "projection.enabled" = "true",
    "projection.bucket_id.type" = "integer",
    "projection.bucket_id.range" = "1,1000000"
)
```

A corresponding SELECT query uses a mapping function in the WHERE clause, as in the following example.

```sql
SELECT
col1,
col2,...,
identifier
```
FROM table
WHERE bucket_id = map_identifier_to_bucket("ID-IN-QUESTION")
AND identifier = "ID-IN-QUESTION"

Replace the `map_identifier_to_bucket` function in the example with any scalar expression that maps an identifier to an integer. For example, the expression could be a simple hash or modulus. The function enforces a constant upper bound on the number of partitions that can ever be projected on the specified dimension. When paired with a file format that supports predicate pushdown such as Apache Parquet or ORC, the bucket technique provides good performance.

For information about writing your own user-defined function like the scalar bucketing function in the preceding example, see Querying with user defined functions (p. 434).

### Amazon Kinesis Data Firehose example

When you use Kinesis Data Firehose to deliver data to Amazon S3, the default configuration writes objects with keys that look like the following example:

```
s3://bucket/prefix/yyyy/MM/dd/HH/file.extension
```

To create an Athena table that finds the partitions automatically at query time, instead of having to add them to the AWS Glue Data Catalog as new data arrives, you can use partition projection.

The following `CREATE TABLE` example uses the default Kinesis Data Firehose configuration.

```sql
CREATE EXTERNAL TABLE my_ingested_data (
    ...
) PARTITIONED BY (datehour STRING)
LOCATION "s3://DOC-EXAMPLE-BUCKET/prefix/"
TBLPROPERTIES ("projection.enabled" = "true",
    "projection.datehour.type" = "date",
    "projection.datehour.format" = "yyyy/MM/dd/HH",
    "projection.datehour.range" = "2021/01/01/00,NOW",
    "projection.datehour.interval" = "1",
    "projection.datehour.interval.unit" = "HOURS",
    "storage.location.template" = "s3://DOC-EXAMPLE-BUCKET/prefix/${datehour}/")
```

The `TBLPROPERTIES` clause in the `CREATE TABLE` statement tells Athena the following:

- Use partition projection when querying the table
- The partition key `datehour` is of type date (which includes an optional time)
- How the dates are formatted
- The range of date times
- Where to find the data on Amazon S3.

When you query the table, Athena calculates the values for `datehour` and uses the storage location template to generate a list of partition locations.
Using the date type

When you use the date type for a projected partition key, you must specify a range. Because you have no data for dates before the Kinesis Data Firehose delivery stream was created, you can use the date of creation as the start. And because you do not have data for dates in the future, you can use the special token NOW as the end.

In the CREATE TABLE example, the start date is specified as January 1, 2021 at midnight UTC.

**Note**
Configure a range that matches your data as closely as possible so that Athena looks only for existing partitions.

When a query is run on the sample table, Athena uses the conditions on the datehour partition key in combination with the range to generate values. Consider the following query:

```
SELECT * FROM my_ingested_data
WHERE datehour >= '2020/12/15/00'
AND datehour < '2021/02/03/15'
```

The first condition in the SELECT query uses a date that is before the start of the date range specified by the CREATE TABLE statement. Because the partition projection configuration specifies no partitions for dates before January 1, 2021, Athena looks for data only in the following locations, and ignores the earlier dates in the query.

```
s3://bucket/prefix/2021/01/01/00/
s3://bucket/prefix/2021/01/01/01/
s3://bucket/prefix/2021/01/01/02/
...
s3://bucket/prefix/2021/02/03/12/
s3://bucket/prefix/2021/02/03/13/
s3://bucket/prefix/2021/02/03/14/
```

Similarly, if the query ran at a date and time before February 3, 2021 at 15:00, the last partition would reflect the current date and time, not the date and time in the query condition.

If you want to query for the most recent data, you can take advantage of the fact that Athena does not generate future dates and specify only a beginning datehour, as in the following example.

```
SELECT * FROM my_ingested_data
WHERE datehour >= '2021/11/09/00'
```

Choosing partition keys

You can specify how partition projection maps the partition locations to partition keys. In the CREATE TABLE example, the date and hour were combined into one partition key called datehour, but other schemes are possible. For example, you could also configure a table with separate partition keys for the year, month, day, and hour.

Queries can be easier to read when there are separate conditions on the day and hour. With that in mind, the following CREATE TABLE example separates the date from the hour.

```
CREATE EXTERNAL TABLE my_ingested_data2 (...
```
PARTITIONED BY (
   day STRING,
   hour INT
)
LOCATION "s3://DOC-EXAMPLE-BUCKET/prefix/"

TBLPROPERTIES (  
   "projection.enabled" = "true",
   "projection.day.type" = "date",
   "projection.day.format" = "yyyy/MM/dd",
   "projection.day.range" = "2021/01/01,NOW",
   "projection.day.interval" = "1",
   "projection.day.interval.unit" = "DAYS",
   "projection.hour.type" = "integer",
   "projection.hour.range" = "0,23",
   "projection.hour.digits" = "2",
   "storage.location.template" = "s3://DOC-EXAMPLE-BUCKET/prefix/${day}/${hour}/"
)

In the example CREATE TABLE statement, the hour is a separate partition key, configured as an integer. The configuration for the hour partition key specifies the range 0 to 23, and that the hour should be formatted with two digits when Athena generates the partition locations.

A query for the my_ingested_data2 table might look like this:

```sql
SELECT *
FROM my_ingested_data2
WHERE day = '2021/11/09'
AND hour > 3
```

**Partition key types and partition projection types**

Note that datehour key in the first CREATE TABLE example is configured as date in the partition projection configuration, but the type of the partition key is string. The same is true for day in the second example. The types in the partition projection configuration only tell Athena how to format the values when it generates the partition locations. The types that you specify do not change the type of the partition key — in queries, datehour and day are of type string.

When a query includes a condition like `day = '2021/11/09'`, Athena parses the string on the right side of the expression using the date format specified in the partition projection configuration. After Athena verifies that the date is within the configured range, it uses the date format again to insert the date as a string into the storage location template.

Similarly, for a query condition like `day > '2021/11/09'`, Athena parses the right side and generates a list of all matching dates within the configured range. It then uses the date format to insert each date into the storage location template to create the list of partition locations.

Writing the same condition as `day > '2021-11-09'` or `day > DATE '2021-11-09'` does not work. In the first case, the date format does not match (note the hyphens instead of the forward slashes), and in the second case, the data types do not match.

**Using custom prefixes and dynamic partitioning**

Kinesis Data Firehose can be configured with custom prefixes and dynamic partitioning. Using these features, you can configure the Amazon S3 keys and set up partitioning schemes that better support your use case. You can also use partition projection with these partitioning schemes and configure them accordingly.

For example, you could use the custom prefix feature to get Amazon S3 keys that have ISO formatted dates instead of the default `yyyy/MM/dd/HH` scheme.
You can also combine custom prefixes with dynamic partitioning to extract a property like `customer_id` from Kinesis Data Firehose messages, as in the following example.

```sql
(prefix/!{timestamp:yyyy}-!{timestamp:MM}-!{timestamp:dd}/!
{partitionKeyFromQuery:customer_id})
```

With that Amazon S3 prefix, the Kinesis Data Firehose delivery stream would write objects to keys such as `s3://bucket/prefix/2021-11-01/customer-1234/file.extension`. For a property like `customer_id`, where the values may not be known in advance, you can use the partition projection type `injected` (p. 275) and use a `CREATE TABLE` statement like the following:

```sql
CREATE EXTERNAL TABLE my_ingested_data3 (  
  ...  
)  
PARTITIONED BY (  
  day STRING,  
  customer_id STRING  
)  
LOCATION "s3://DOC-EXAMPLE-BUCKET/prefix/"  
TBLPROPERTIES (  
  "projection.enabled" = "true",  
  "projection.day.type" = "date",  
  "projection.day.format" = "yyyy-MM-dd",  
  "projection.day.range" = "2021-01-01,NOW",  
  "projection.day.interval" = "1",  
  "projection.day.interval.unit" = "DAYS",  
  "projection.customer_id.type" = "injected",  
  "storage.location.template" = "s3://DOC-EXAMPLE-BUCKET/prefix/${day}/${customer_id}/"  
)
```

When you query a table that has a partition key of type `injected`, your query must include a value for that partition key. A query for the `my_ingested_data3` table might look like this:

```sql
SELECT *  
FROM my_ingested_data3  
WHERE day BETWEEN '2021-11-01' AND '2021-11-30'  
AND customer_id = 'customer-1234'
```

**ISO formatted dates**

Because the values for the `day` partition key are ISO formatted, you can also use the `DATE` type for the `day` partition key instead of `STRING`, as in the following example:

```sql
PARTITIONED BY (day DATE, customer_id STRING)
```

When you query, this strategy allows you to use date functions on the partition key without parsing or casting, as in the following example:

```sql
SELECT *  
FROM my_ingested_data3  
WHERE day > CURRENT_DATE - INTERVAL '7' DAY  
AND customer_id = 'customer-1234'
```
Creating a table from query results (CTAS)

A CREATE TABLE AS SELECT (CTAS) query creates a new table in Athena from the results of a SELECT statement from another query. Athena stores data files created by the CTAS statement in a specified location in Amazon S3. For syntax, see CREATE TABLE AS (p. 744).

Use CTAS queries to:

- Create tables from query results in one step, without repeatedly querying raw data sets. This makes it easier to work with raw data sets.
- Transform query results and migrate tables into other table formats such as Apache Iceberg. This improves query performance and reduces query costs in Athena. For information, see Creating Iceberg tables (p. 504).
- Transform query results into storage formats such as Parquet and ORC. This improves query performance and reduces query costs in Athena. For information, see Columnar storage formats (p. 259).
- Create copies of existing tables that contain only the data you need.

Topics

- Considerations and limitations for CTAS queries (p. 281)
- Running CTAS queries in the console (p. 283)
- Bucketing vs partitioning (p. 284)
- Examples of CTAS queries (p. 286)
- Using CTAS and INSERT INTO for ETL and data analysis (p. 290)
- Using CTAS and INSERT INTO to create a table with more than 100 partitions (p. 295)

Considerations and limitations for CTAS queries

The following table describes what you need to know about CTAS queries in Athena:

<table>
<thead>
<tr>
<th>Item</th>
<th>What you need to know</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTAS query syntax</td>
<td>The CTAS query syntax differs from the syntax of CREATE [EXTERNAL] TABLE used for creating tables. See CREATE TABLE AS (p. 744).</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong></td>
</tr>
<tr>
<td></td>
<td>Table, database, or column names for CTAS queries should not contain quotes or backticks. To ensure this, check that your table, database, or column names do not represent reserved words (p. 256), and do not contain special characters (which require enclosing them in quotes or backticks). For more information, see Names for tables, databases, and columns (p. 254).</td>
</tr>
<tr>
<td>CTAS queries vs views</td>
<td>CTAS queries write new data to a specified location in Amazon S3, whereas views do not write any data.</td>
</tr>
<tr>
<td>Location of CTAS query results</td>
<td>If your workgroup overrides the client-side setting (p. 638) for query results location, Athena creates your table in the location s3://&lt;workgroup-query-</td>
</tr>
</tbody>
</table>
### Considerations and limitations for CTAS queries

<table>
<thead>
<tr>
<th>Item</th>
<th>What you need to know</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>results-location</strong></td>
<td>To see the query results location specified for the workgroup, view the workgroup's details (p. 643).</td>
</tr>
<tr>
<td></td>
<td>If your workgroup does not override the query results location, you can use the syntax WITH (external_location = 's3://location/') in your CTAS query to specify where your CTAS query results are stored.</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong></td>
</tr>
<tr>
<td></td>
<td>The external_location property must specify a location that is empty. A CTAS query checks that the path location (prefix) in the bucket is empty and never overwrites the data if the location already has data in it. To use the same location again, delete the data in the key prefix location in the bucket.</td>
</tr>
</tbody>
</table>
|   | If you omit the external_location syntax and are not using the workgroup setting, Athena uses your client-side setting (p. 344) for the query results location and creates your table in the location s3://<client-query-results-location>/<Unsaved-or-query-name>/<year>/<month/<date>/tables/<query-id>/.

### Locating Orphaned Files

If a CTAS or INSERT INTO statement fails, it is possible that orphaned data are left in the data location. Because Athena does not delete any data (even partial data) from your bucket, you might be able to read this partial data in subsequent queries. To locate orphaned files for inspection or deletion, you can use the data manifest file that Athena provides to track the list of files to be written. For more information, see Identifying query output files (p. 352) and DataManifestLocation.

### Formats for storing query results

The results of CTAS queries are stored in Parquet by default if you don't specify a data storage format. You can store CTAS results in PARQUET, ORC, AVRO, JSON, and TEXTFILE. Multi-character delimiters are not supported for the CTAS TEXTFILE format. CTAS queries do not require specifying a SerDe to interpret format transformations. See Example: Writing Query Results to a Different Format (p. 287).

### Compression formats

GZIP compression is used for CTAS query results in JSON and TEXTFILE formats. For Parquet, you can use GZIP or SNAPPY, and the default is GZIP. For ORC, you can use LZ4, SNAPPY, ZLIB, or ZSTD, and the default is ZLIB. For CTAS examples that specify compression, see Example: Specifying Data Storage and Compression Formats (p. 287).

### Partition and Bucket Limits

You can partition and bucket the results data of a CTAS query. For more information, see Bucketing vs partitioning (p. 284). Athena supports writing to 100 unique partition and bucket combinations. For example, if no buckets are defined in the destination table, you can specify a maximum of 100 partitions. If you specify five buckets, 20 partitions (each with five buckets) are allowed. If you exceed this count, an error occurs.

Include partitioning and bucketing predicates at the end of the WITH clause that specifies properties of the destination table. For more information, see Example: Creating Bucketed and Partitioned Tables (p. 289) and Bucketing vs partitioning (p. 284).

For information about working around the 100-partition limitation, see Using CTAS and INSERT INTO to create a table with more than 100 partitions (p. 295).

### Encryption

You can encrypt CTAS query results in Amazon S3, similar to the way you encrypt other query results in Athena. For more information, see Encrypting Athena query results stored in Amazon S3 (p. 523).
Running CTAS queries in the console

In the Athena console, you can create a CTAS query from another query.

To create a CTAS query from another query

1. Run the query in the Athena console query editor.
2. At the bottom of the query editor, choose the Create option, and then choose Table from query.
3. In the Create table as select form, complete the fields as follows:
   a. For Table name, enter the name for your new table. Use only lowercase and underscores, such as my_select_query_parquet.
   b. For Database configuration, use the options to choose an existing database or create a database.
   c. (Optional) In Result configuration, for Location of CTAS query results, if your workgroup query results location setting does not override this option, do one of the following:
      • Enter the path to an existing S3 location in the search box, or choose Browse S3 to choose a location from a list.
      • Choose View to open the Buckets page of the Amazon S3 console where you can view more information about your existing buckets and choose or create a bucket with your own settings.

   You should specify an empty location in Amazon S3 where the data will be output. If data already exists in the location that you specify, the query fails with an error.

   If your workgroup query results location setting overrides this location setting, Athena creates your table in the location s3://workgroup_query_results_location/tables/query_id/
   d. For Data format, specify the format that your data is in.
      • Table type – The default table type in Athena is Apache Hive.
      • File format – Choose among options like CSV, TSV, JSON, Parquet, or ORC. For information about the Parquet and ORC formats, see Columnar storage formats (p. 259).
      • Write compression – (Optional) Choose a compression format. Athena supports a variety of compression formats for reading and writing data, including reading from a table that uses multiple compression formats. For example, Athena can successfully read the data in a table that uses Parquet file format when some Parquet files are compressed with Snappy and other Parquet files are compressed with GZIP. The same principle applies for ORC, text file, and JSON storage formats. For more information, see Athena compression support (p. 299).
      • Partitions – (Optional) Select the columns that you want to partition. Partitioning your data restricts the amount of data scanned by each query, thus improving performance and
Bucketing vs partitioning

You can specify partitioning and bucketing, for storing data from CTAS query results in Amazon S3. For information about CTAS queries, see CREATE TABLE AS SELECT (CTAS) (p. 281).

This section discusses partitioning and bucketing as they apply to CTAS queries only. For general guidelines about using partitioning in CREATE TABLE queries, see Top performance tuning tips for Amazon Athena.

Use the following tips to decide whether to partition and/or to configure bucketing, and to select columns in your CTAS queries by which to do so:

- **Partitioning CTAS query results** works well when the number of partitions you plan to have is limited. When you run a CTAS query, Athena writes the results to a specified location in Amazon S3. If you specify partitions, it creates them and stores each partition in a separate partition folder in the same location. The maximum number of partitions you can configure with CTAS query results in one query is 100. However, you can work around this limitation. For more information, see Using CTAS and INSERT INTO to create a table with more than 100 partitions (p. 295).

Having partitions in Amazon S3 helps with Athena query performance, because this helps you run targeted queries for only specific partitions. Athena then scans only those partitions, saving you query costs and query time. For information about partitioning syntax, search for partitioned_by in CREATE TABLE AS (p. 744).

Partition data by those columns that have similar characteristics, such as records from the same department, and that can have a limited number of possible values, such as a limited number of distinct departments in an organization. This characteristic is known as **data cardinality**. For example, if you partition by the column department, and this column has a limited number of distinct values, partitioning by department works well and decreases query latency.

- **Bucketing CTAS query results** works well when you bucket data by the column that has high cardinality and evenly distributed values.
For example, columns storing timestamp data could potentially have a very large number of distinct values, and their data is evenly distributed across the data set. This means that a column storing timestamp type data will most likely have values and won't have nulls. This also means that data from such a column can be put in many buckets, where each bucket will have roughly the same amount of data stored in Amazon S3.

To choose the column by which to bucket the CTAS query results, use the column that has a high number of values (high cardinality) and whose data can be split for storage into many buckets that will have roughly the same amount of data. Columns that are sparsely populated with values are not good candidates for bucketing. This is because you will end up with buckets that have less data and other buckets that have a lot of data. By comparison, columns that you predict will almost always have values, such as timestamp type values, are good candidates for bucketing. This is because their data has high cardinality and can be stored in roughly equal chunks.

For more information about bucketing syntax, search for bucketed_by in \texttt{CREATE TABLE AS} (p. 744).

To conclude, you can partition and use bucketing for storing results of the same CTAS query. These techniques for writing data do not exclude each other. Typically, the columns you use for bucketing differ from those you use for partitioning.

For example, if your dataset has columns department, sales_quarter, and customer_id (integer type), you can partition your CTAS query results by department and sales_quarter. These columns have relatively low cardinality of values: a limited number of departments and sales quarters. Also, for partitions, it does not matter if some records in your dataset have null or no values assigned for these columns. What matters is that data with the same characteristics, such as data from the same department, will be in one partition that you can query in Athena.

At the same time, because all of your data has integer type values stored in the column customer_id, you can configure bucketing for the same query results by the column customer_id. This column has high cardinality. You can store its data in more than one bucket in Amazon S3. Consider an opposite scenario: if you don't create buckets for the column customer_id and run a query for particular customer ID values, then you would have to scan a very large amount of data stored in a single location in Amazon S3. Instead, if you configure buckets for storing your ID-related results, you can only scan and query buckets that have your value and avoid long-running queries that scan a large amount of data.

### Data types supported for filtering on bucketed columns

You can reduce the amount of data scanned by adding filters on bucketed columns that have certain data types. Athena supports such filtering only on bucketed columns with the following data types:

- TINYINT
- SMALLINT
- INT
- BIGINT
- BOOLEAN
- STRING
- TIMESTAMP
Examples of CTAS queries

Use the following examples to create CTAS queries. For information about the CTAS syntax, see CREATE TABLE AS (p. 744).

In this section:
- Example: Duplicating a Table by Selecting All Columns (p. 286)
- Example: Selecting Specific Columns From One or More Tables (p. 286)
- Example: Creating an Empty Copy of an Existing Table (p. 286)
- Example: Specifying Data Storage and Compression Formats (p. 287)
- Example: Writing Query Results to a Different Format (p. 287)
- Example: Creating Unpartitioned Tables (p. 287)
- Example: Creating Partitioned Tables (p. 288)
- Example: Creating Bucketed and Partitioned Tables (p. 289)
- Example: Creating an Iceberg table with Parquet data (p. 289)
- Example: Creating an Iceberg table with Avro data (p. 289)

Example Example: Duplicating a table by selecting all columns

The following example creates a table by copying all columns from a table:

```sql
CREATE TABLE new_table AS
SELECT * 
FROM old_table;
```

In the following variation of the same example, your SELECT statement also includes a WHERE clause. In this case, the query selects only those rows from the table that satisfy the WHERE clause:

```sql
CREATE TABLE new_table AS
SELECT *
FROM old_table
WHERE condition;
```

Example Example: Selecting specific columns from one or more tables

The following example creates a new query that runs on a set of columns from another table:

```sql
CREATE TABLE new_table AS
SELECT column_1, column_2, ... column_n 
FROM old_table;
```

This variation of the same example creates a new table from specific columns from multiple tables:

```sql
CREATE TABLE new_table AS
SELECT column_1, column_2, ... column_n 
FROM old_table_1, old_table_2, ... old_table_n;
```

Example Example: Creating an empty copy of an existing table

The following example uses WITH NO DATA to create a new table that is empty and has the same schema as the original table:

```sql
CREATE TABLE new_table
```
Example: Specifying data storage and compression formats

With CTAS, you can use a source table in one storage format to create another table in a different storage format.

Use the `format` property to specify ORC, PARQUET, AVRO, JSON, or TEXTFILE as the storage format for the new table.

For the PARQUET, ORC, TEXTFILE, and JSON storage formats, use the `write_compression` property to specify the compression format for the new table's data. For information about the compression formats that each file format supports, see Athena compression support (p. 299).

The following example specifies that data in the table `new_table` be stored in Parquet format and use Snappy compression. The default compression for Parquet is GZIP.

```sql
CREATE TABLE new_table
WITH (format = 'Parquet',
     write_compression = 'SNAPPY')
AS SELECT *
FROM old_table;
```

The following example specifies that data in the table `new_table` be stored in ORC format using Snappy compression. The default compression for ORC is ZLIB.

```sql
CREATE TABLE new_table
WITH (format = 'ORC',
     write_compression = 'SNAPPY')
AS SELECT *
FROM old_table;
```

The following example specifies that data in the table `new_table` be stored in textfile format using Snappy compression. The default compression for both the textfile and JSON formats is GZIP.

```sql
CREATE TABLE new_table
WITH (format = 'TEXTFILE',
     write_compression = 'SNAPPY')
AS SELECT *
FROM old_table;
```

Example: Writing query results to a different format

The following CTAS query selects all records from `old_table`, which could be stored in CSV or another format, and creates a new table with underlying data saved to Amazon S3 in ORC format:

```sql
CREATE TABLE my_orc_ctas_table
WITH (external_location = 's3://my_athena_results/my_orc_stas_table/',
     format = 'ORC')
AS SELECT *
FROM old_table;
```

Example: Creating unpartitioned tables

The following examples create tables that are not partitioned. The table data is stored in different formats. Some of these examples specify the external location.
The following example creates a CTAS query that stores the results as a text file:

```sql
CREATE TABLE ctas_csv_unpartitioned
WITH (format = 'TEXTFILE',
      external_location = 's3://my_athena_results/ctas_csv_unpartitioned/')
AS SELECT key1, name1, address1, comment1
FROM table1;
```

In the following example, results are stored in Parquet, and the default results location is used:

```sql
CREATE TABLE ctas_parquet_unpartitioned
WITH (format = 'PARQUET')
AS SELECT key1, name1, comment1
FROM table1;
```

In the following query, the table is stored in JSON, and specific columns are selected from the original table's results:

```sql
CREATE TABLE ctas_json_unpartitioned
WITH (format = 'JSON',
      external_location = 's3://my_athena_results/ctas_json_unpartitioned/')
AS SELECT key1, name1, address1, comment1
FROM table1;
```

In the following example, the format is ORC:

```sql
CREATE TABLE ctas_orc_unpartitioned
WITH (format = 'ORC')
AS SELECT key1, name1, comment1
FROM table1;
```

In the following example, the format is Avro:

```sql
CREATE TABLE ctas_avro_unpartitioned
WITH (format = 'AVRO',
      external_location = 's3://my_athena_results/ctas_avro_unpartitioned/')
AS SELECT key1, name1, comment1
FROM table1;
```

Example Example: Creating partitioned tables

The following examples show CREATE TABLE AS SELECT queries for partitioned tables in different storage formats, using partitioned_by, and other properties in the WITH clause. For syntax, see CTAS table properties (p. 745). For more information about choosing the columns for partitioning, see Bucketing vs partitioning (p. 284).

Note
List partition columns at the end of the list of columns in the SELECT statement. You can partition by more than one column, and have up to 100 unique partition and bucket combinations. For example, you can have 100 partitions if no buckets are specified.

```sql
CREATE TABLE ctas_csv_partitioned
WITH (format = 'TEXTFILE',
      external_location = 's3://my_athena_results/ctas_csv_partitioned/'),
```
CREATE TABLE ctas_json_partitioned
WITH (format = 'JSON',
      external_location = 's3://my_athena_results/ctas_json_partitioned/',
      partitioned_by = ARRAY['key1'])
AS select name1, address1, comment1, key1
FROM table1;

Example Example: Creating bucketed and partitioned tables

The following example shows a CREATE TABLE AS SELECT query that uses both partitioning and bucketing for storing query results in Amazon S3. The table results are partitioned and bucketed by different columns. Athena supports a maximum of 100 unique bucket and partition combinations. For example, if you create a table with five buckets, 20 partitions with five buckets each are supported. For syntax, see CTAS table properties (p. 745).

For information about choosing the columns for bucketing, see Bucketing vs partitioning (p. 284).

CREATE TABLE ctas_avro_bucketed
WITH (format = 'AVRO',
      external_location = 's3://my_athena_results/ctas_avro_bucketed/',
      partitioned_by = ARRAY['nationkey'],
      bucketed_by = ARRAY['mktsegment'],
      bucket_count = 3)
AS SELECT key1, name1, address1, phone1, acctbal, mktsegment, comment1, nationkey
FROM table1;

Example Example: Creating an Iceberg table with Parquet data

The following example creates an Iceberg table with Parquet data files. The files are partitioned by a month transform on the dt column in table1. The example updates the retention properties on the table so that 10 snapshots are retained by default on every branch in the table. Snapshots within the past 30 days are also retained.

CREATE TABLE ctas_iceberg_parquet
WITH (table_type = 'ICEBERG',
      format = 'PARQUET',
      location = 's3://my_athena_results/ctas_iceberg_parquet/',
      is_external = false,
      partitioning = ARRAY['month(dt)'],
      vacuum_min_snapshots_to_keep = 10,
      vacuum_max_snapshot_age_ms = 259200 )
AS SELECT key1, name1, dt FROM table1;

Example Example: Creating an Iceberg table with Avro data

The following example creates an Iceberg table with Avro data files partitioned by key1.

CREATE TABLE ctas_iceberg_avro
WITH ( format = 'AVRO',
      location = 's3://my_athena_results/ctas_iceberg_avro/',
      is_external = false,
      table_type = 'ICEBERG',
      partitioned_by = ARRAY['key1'])
AS SELECT name1, address1, comment1, key1
FROM tables1;
Using CTAS and INSERT INTO for ETL and data analysis

You can use Create Table as Select (CTAS (p. 281)) and INSERT INTO (p. 697) statements in Athena to extract, transform, and load (ETL) data into Amazon S3 for data processing. This topic shows you how to use these statements to partition and convert a dataset into columnar data format to optimize it for data analysis.

CTAS statements use standard SELECT (p. 692) queries to create new tables. You can use a CTAS statement to create a subset of your data for analysis. In one CTAS statement, you can partition the data, specify compression, and convert the data into a columnar format like Apache Parquet or Apache ORC. When you run the CTAS query, the tables and partitions that it creates are automatically added to the AWS Glue Data Catalog. This makes the new tables and partitions that it creates immediately available for subsequent queries.

INSERT INTO statements insert new rows into a destination table based on a SELECT query statement that runs on a source table. You can use INSERT INTO statements to transform and load source table data in CSV format into destination table data using all transforms that CTAS supports.

Overview

In Athena, use a CTAS statement to perform an initial batch conversion of the data. Then use multiple INSERT INTO statements to make incremental updates to the table created by the CTAS statement.

Steps

- Step 1: Create a table based on the original dataset (p. 290)
- Step 2: Use CTAS to partition, convert, and compress the data (p. 291)
- Step 3: Use INSERT INTO to add data (p. 292)
- Step 4: Measure performance and cost differences (p. 294)

Step 1: Create a table based on the original dataset

The example in this topic uses an Amazon S3 readable subset of the publicly available NOAA global historical climatology network daily (GHCN-d) dataset. The data on Amazon S3 has the following characteristics.

<table>
<thead>
<tr>
<th>Location: s3://aws-bigdata-blog/artifacts/athena-ctas-insert-into-blog/</th>
<th>Total objects: 41727</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of CSV dataset: 11.3 GB</td>
<td></td>
</tr>
<tr>
<td>Region: us-east-1</td>
<td></td>
</tr>
</tbody>
</table>

The original data is stored in Amazon S3 with no partitions. The data is in CSV format in files like the following.

| 2019-10-31 13:06:57 | 413.1 KiB | artifacts/athena-ctas-insert-into-blog/2010.csv0000 |
| 2019-10-31 13:06:57 | 34.4 KiB | artifacts/athena-ctas-insert-into-blog/2010.csv0002 |
Step 2: Use CTAS to partition, convert, and compress the data

The file sizes in this sample are relatively small. By merging them into larger files, you can reduce the total number of files, enabling better query performance. You can use CTAS and INSERT INTO statements to enhance query performance.

To create a database and table based on the sample dataset

1. In the Athena console, choose the US East (N. Virginia) AWS Region. Be sure to run all queries in this tutorial in us-east-1.
2. In the Athena query editor, run the CREATE DATABASE (p. 739) command to create a database.

```
CREATE DATABASE blogdb
```

3. Run the following statement to create a table (p. 740).

```
CREATE EXTERNAL TABLE `blogdb`.`original_csv` (  
`id` string,  
`date` string,  
`element` string,  
`datavalue` bigint,  
`mflag` string,  
`qflag` string,  
`sflag` string,  
`obstime` bigint)  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY ','  
STORED AS INPUTFORMAT  
'org.apache.hadoop.mapred.TextInputFormat'  
OUTPUTFORMAT  
'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'  
LOCATION  
's3://aws-bigdata-blog/artifacts/athena-ctas-insert-into-blog/'
```

Step 2: Use CTAS to partition, convert, and compress the data

After you create a table, you can use a single CTAS (p. 281) statement to convert the data to Parquet format with Snappy compression and to partition the data by year.

The table you created in Step 1 has a date field with the date formatted as YYYYMMDD (for example, 20100104). Because the new table will be partitioned on year, the sample statement in the following procedure uses the Presto function `substr("date",1,4)` to extract the year value from the date field.

To convert the data to parquet format with snappy compression, partitioning by year

- Run the following CTAS statement, replacing `your-bucket` with your Amazon S3 bucket location.

```
CREATE table new_parquet  
WITH (format='PARQUET',  
parquet_compression='SNAPPY',  
partitioned_by=array['year'],  
external_location = 's3://your-bucket/optimized-data/')  
AS  
SELECT id,  
    date,
```
Step 3: Use INSERT INTO to add data

In Step 2, you used CTAS to create a table with partitions for the years 2015 to 2019. However, the original dataset also contains data for the years 2010 to 2014. Now you add that data using an INSERT INTO (p. 697) statement.

```
element, datavalue, mflag, qflag, sflag, obstime, 
substr("date",1,4) AS year
FROM original_csv
WHERE cast(substr("date",1,4) AS bigint) >= 2015
    AND cast(substr("date",1,4) AS bigint) <= 2019
```

Note
In this example, the table that you create includes only the data from 2015 to 2019. In Step 3, you add new data to this table using the INSERT INTO command.

When the query completes, use the following procedure to verify the output in the Amazon S3 location that you specified in the CTAS statement.

To see the partitions and parquet files created by the CTAS statement

1. To show the partitions created, run the following AWS CLI command. Be sure to include the final forward slash (/).

   ```
   aws s3 ls s3://your-bucket/optimized-data/
   ```

   The output shows the partitions.

   ```
   PRE year=2015/
   PRE year=2016/
   PRE year=2017/
   PRE year=2018/
   PRE year=2019/
   ```

2. To see the Parquet files, run the following command. Note that the | head -5 option, which restricts the output to the first five results, is not available on Windows.

   ```
   aws s3 ls s3://your-bucket/optimized-data/ --recursive --human-readable | head -5
   ```

   The output resembles the following.

   ```
   2019-10-31 14:51:05   7.3 MiB optimized-data/
   year=2015/20191031_215021_00001_3f42d_1be48df2-3154-438b-b61d-8fb23809679d
   2019-10-31 14:51:05   7.0 MiB optimized-data/
   year=2015/20191031_215021_00001_3f42d_2a57f4e2-ffa0-4be3-9c3f-28b16d86ed5a
   2019-10-31 14:51:05   9.9 MiB optimized-data/
   year=2015/20191031_215021_00001_3f42d_34381db1-00ca-4092-bd65-ab04e06dc799
   2019-10-31 14:51:05   7.5 MiB optimized-data/
   year=2015/20191031_215021_00001_3f42d_354a2bc1-345f-4996-9073-096cb863088d
   2019-10-31 14:51:05   6.9 MiB optimized-data/
   year=2015/20191031_215021_00001_3f42d_42da4cfd-6e21-40a1-8152-0b902da585a1
   ```

Step 3: Use INSERT INTO to add data

In Step 2, you used CTAS to create a table with partitions for the years 2015 to 2019. However, the original dataset also contains data for the years 2010 to 2014. Now you add that data using an INSERT INTO (p. 697) statement.
To add data to the table using one or more INSERT INTO statements

1. Run the following INSERT INTO command, specifying the years before 2015 in the WHERE clause.

```
INSERT INTO new_parquet
SELECT id,
     date,
     element,
     datavalue,
     mflag,
     qflag,
     sflag,
     obstime,
     substr("date",1,4) AS year
FROM original_csv
WHERE cast(substr("date",1,4) AS bigint) < 2015
```

2. Run the `aws s3 ls` command again, using the following syntax.

```
aws s3 ls s3://your-bucket/optimized-data/
```

The output shows the new partitions.

```
PRE year=2010/
PRE year=2011/
PRE year=2012/
PRE year=2013/
PRE year=2014/
PRE year=2015/
PRE year=2016/
PRE year=2017/
PRE year=2018/
PRE year=2019/
```

3. To see the reduction in the size of the dataset obtained by using compression and columnar storage in Parquet format, run the following command.

```
aws s3 ls s3://your-bucket/optimized-data/ --recursive --human-readable --summarize
```

The following results show that the size of the dataset after parquet with Snappy compression is 1.2 GB.

```
... 2020-01-22 18:12:02 2.8 MiB optimized-data/
year=2019/20200122_181132_00003_nja5r_f0182e6c-38f4-4245-afa2-9f5bfa8d6d8f
2020-01-22 18:11:59 3.7 MiB optimized-data/
year=2019/20200122_181132_00005_nja5r_fd9906b7-06cf-4055-a05b-f050e139946e
Total Objects: 500
   Total Size: 1.2 GiB
```

4. If more CSV data is added to original table, you can add that data to the parquet table by using INSERT INTO statements. For example, if you had new data for the year 2020, you could run the following INSERT INTO statement. The statement adds the data and the relevant partition to the new_parquet table.

```
INSERT INTO new_parquet
SELECT id,
     date,
     element,
```
Step 4: Measure performance and cost differences

After you transform the data, you can measure the performance gains and cost savings by running the same queries on the new and old tables and comparing the results.

Note
For Athena per-query cost information, see Amazon Athena pricing.

To measure performance gains and cost differences

1. Run the following query on the original table. The query finds the number of distinct IDs for every value of the year.

   ```sql
   SELECT substr("date",1,4) as year,
       COUNT(DISTINCT id)
   FROM original_csv
   GROUP BY 1 ORDER BY 1 DESC
   ```

2. Note the time that the query ran and the amount of data scanned.
3. Run the same query on the new table, noting the query runtime and amount of data scanned.

   ```sql
   SELECT year,
       COUNT(DISTINCT id)
   FROM new_parquet
   GROUP BY 1 ORDER BY 1 DESC
   ```

4. Compare the results and calculate the performance and cost difference. The following sample results show that the test query on the new table was faster and cheaper than the query on the old table.

<table>
<thead>
<tr>
<th>Table</th>
<th>Runtime</th>
<th>Data scanned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>16.88 s</td>
<td>11.35 GB</td>
</tr>
<tr>
<td>New</td>
<td>3.79 s</td>
<td>428.05 MB</td>
</tr>
</tbody>
</table>

5. Run the following sample query on the original table. The query calculates the average maximum temperature (Celsius), average minimum temperature (Celsius), and average rainfall (mm) for the Earth in 2018.

   ```sql
   SELECT element, round(avg(CAST(datavalue AS real)/10),2) AS value
   FROM original_csv
   WHERE element IN ('TMIN', 'TMAX', 'PRCP')
   GROUP BY element
   ```
Note the time that the query ran and the amount of data scanned.

6. Run the same query on the new table, noting the query runtime and amount of data scanned.

    SELECT element, round(avg(CAST(datavalue AS real)/10),2) AS value
    FROM new_parquet
    WHERE element IN ('TMIN', 'TMAX', 'PRCP') and year = '2018'
    GROUP BY 1

7. Compare the results and calculate the performance and cost difference. The following sample results show that the test query on the new table was faster and cheaper than the query on the old table.

<table>
<thead>
<tr>
<th>Table</th>
<th>Runtime</th>
<th>Data scanned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>18.65 seconds</td>
<td>11.35 GB</td>
</tr>
<tr>
<td>New</td>
<td>1.92 seconds</td>
<td>68 MB</td>
</tr>
</tbody>
</table>

Summary

This topic showed you how to perform ETL operations using CTAS and INSERT INTO statements in Athena. You performed the first set of transformations using a CTAS statement that converted data to the Parquet format with Snappy compression. The CTAS statement also converted the dataset from non-partitioned to partitioned. This reduced its size and lowered the costs of running the queries. When new data becomes available, you can use an INSERT INTO statement to transform and load the data into the table that you created with the CTAS statement.

Using CTAS and INSERT INTO to create a table with more than 100 partitions

You can create up to 100 partitions per query with a CREATE TABLE AS SELECT (CTAS (p. 281)) query. Similarly, you can add a maximum of 100 partitions to a destination table with an INSERT INTO statement.

If you exceed this limitation, you may receive the error message HIVE_TOO_MANY_OPEN_PARTITIONS: Exceeded limit of 100 open writers for partitions/buckets. To work around this limitation, you can use a CTAS statement and a series of INSERT INTO statements that create or insert up to 100 partitions each.

The example in this topic uses a database called tpch100 whose data resides in the Amazon S3 bucket location s3://<my-tpch-bucket>/.

To use CTAS and INSERT INTO to create a table of more than 100 partitions

1. Use a CREATE EXTERNAL TABLE statement to create a table partitioned on the field that you want.

   The following example statement partitions the data by the column l_shipdate. The table has 2525 partitions.

   ```sql
   CREATE EXTERNAL TABLE `tpch100.lineitem_parq_partitioned`(
     `l_orderkey` int,
     `l_partkey` int,
   ```
Creating a table with more than 100 partitions

```sql
'l_suppkey' int,
'l_linenumber' int,
'l_quantity' double,
'l_extendedprice' double,
'l_discount' double,
'l_tax' double,
'l_returnflag' string,
'l_linenumber' string,
'l_commitdate' string,
'l_receiptdate' string,
'l_shipinstruct' string,
'l_comment' string)
PARTITIONED BY (l_shipdate string)
ROW FORMAT SERDE 'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat'
LOCATION 's3://<my-tpch-bucket>/lineitem/'
```

2. Run a SHOW PARTITIONS `<table_name>` command like the following to list the partitions.

```
SHOW PARTITIONS lineitem_parq_partitioned
```

Following are partial sample results.

```sql
/*
l_shipdate=1992-01-02
l_shipdate=1992-01-03
l_shipdate=1992-01-04
l_shipdate=1992-01-05
l_shipdate=1992-01-06
...

l_shipdate=1998-11-24
l_shipdate=1998-11-25
l_shipdate=1998-11-26
l_shipdate=1998-11-27
l_shipdate=1998-11-28
l_shipdate=1998-11-29
l_shipdate=1998-11-30
l_shipdate=1998-12-01
*/
```

3. Run a CTAS query to create a partitioned table.

The following example creates a table called `my_lineitem_parq_partitioned` and uses the `WHERE` clause to restrict the `DATE` to earlier than 1992-02-01. Because the sample dataset starts with January 1992, only partitions for January 1992 are created.

```
CREATE table my_lineitem_parq_partitioned
WITH (partitioned_by = ARRAY['l_shipdate']) AS
SELECT l_orderkey,
       l_partkey,
       l_suppkey,
       l_linenumber,
       l_quantity,
       l_extendedprice,
       l_discount,
       l_tax,
       l_returnflag,
```
Creating a table with more than 100 partitions

```sql
CREATE TABLE my_lineitem_parq_partitioned
USING parquet
PARTITIONED BY (l_shipdate)
AS SELECT 
    l_orderkey, 
    l_partkey, 
    l_suppkey, 
    l_linenumber, 
    l_quantity, 
    l_extendedprice, 
    l_discount, 
    l_tax, 
    l_returnflag, 
    l_linestatus, 
FROM tpch100.lineitem_parq_partitioned 
WHERE cast(l_shipdate as timestamp) < DATE ('1992-02-01');
```

4. Run the `SHOW PARTITIONS` command to verify that the table contains the partitions that you want.

```sql
SHOW PARTITIONS my_lineitem_parq_partitioned;
```

The partitions in the example are from January 1992.

```sql
/*
l_shipdate=1992-01-02
l_shipdate=1992-01-03
l_shipdate=1992-01-04
l_shipdate=1992-01-05
l_shipdate=1992-01-06
l_shipdate=1992-01-07
l_shipdate=1992-01-08
l_shipdate=1992-01-09
l_shipdate=1992-01-10
l_shipdate=1992-01-11
l_shipdate=1992-01-12
l_shipdate=1992-01-13
l_shipdate=1992-01-14
l_shipdate=1992-01-15
l_shipdate=1992-01-16
l_shipdate=1992-01-17
l_shipdate=1992-01-18
l_shipdate=1992-01-19
l_shipdate=1992-01-20
l_shipdate=1992-01-21
l_shipdate=1992-01-22
l_shipdate=1992-01-23
l_shipdate=1992-01-24
l_shipdate=1992-01-25
l_shipdate=1992-01-26
l_shipdate=1992-01-27
l_shipdate=1992-01-28
l_shipdate=1992-01-29
l_shipdate=1992-01-30
l_shipdate=1992-01-31
*/
```

5. Use an `INSERT INTO` statement to add partitions to the table.

The following example adds partitions for the dates from the month of February 1992.

```sql
INSERT INTO my_lineitem_parq_partitioned
SELECT 
    l_orderkey, 
    l_partkey, 
    l_suppkey, 
    l_linenumber, 
    l_quantity, 
    l_extendedprice, 
    l_discount, 
    l_tax, 
    l_returnflag, 
    l_linestatus,
FROM tpch100.lineitem_parq_partitioned 
WHERE cast(l_shipdate as timestamp) < DATE ('1992-02-01');
```
l_commitdate,
l_receiptdate,
l_shipinstruct,
l_comment,
l_shipdate
FROM tpch100.lineitem_parq_partitioned
WHERE cast(l_shipdate as timestamp) >= DATE ('1992-02-01')
AND cast(l_shipdate as timestamp) < DATE ('1992-03-01');

6. Run SHOW PARTITIONS again.

```
SHOW PARTITIONS my_lineitem_parq_partitioned;
```

The sample table now has partitions from both January and February 1992.

```
/*
l_shipdate=1992-01-02
l_shipdate=1992-01-03
l_shipdate=1992-01-04
l_shipdate=1992-01-05
l_shipdate=1992-01-06
...
l_shipdate=1992-02-20
l_shipdate=1992-02-21
l_shipdate=1992-02-22
l_shipdate=1992-02-23
l_shipdate=1992-02-24
l_shipdate=1992-02-25
l_shipdate=1992-02-26
l_shipdate=1992-02-27
l_shipdate=1992-02-28
l_shipdate=1992-02-29
*/
```

7. Continue using INSERT INTO statements that read and add no more than 100 partitions each. Continue until you reach the number of partitions that you require.

   **Important**

   When setting the WHERE condition, be sure that the queries don't overlap. Otherwise, some partitions might have duplicated data.
Athena compression support

Athena supports a variety of compression formats for reading and writing data, including reading from a table that uses multiple compression formats. For example, Athena can successfully read the data in a table that uses Parquet file format when some Parquet files are compressed with Snappy and other Parquet files are compressed with GZIP. The same principle applies for ORC, text file, and JSON storage formats.

Athena supports the following compression formats:

- **BZIP2** – Format that uses the Burrows-Wheeler algorithm.
- **DEFLATE** – Compression algorithm based on LZSS and Huffman coding. Deflate is relevant only for the Avro file format.
- **GZIP** – Compression algorithm based on Deflate. GZIP is the default write compression format for files in the Parquet and text file storage formats. Files in the `tar.gz` format are not supported.
- **LZ4** – This member of the Lempel-Ziv 77 (LZ7) family also focuses on compression and decompression speed rather than maximum compression of data. LZ4 has the following framing formats:
  - **LZ4 Raw/Unframed** – An unframed, standard implementation of the LZ4 block compression format. For more information, see the [LZ4 block format description](https://github.com/lz4/lz4) on GitHub.
  - **LZ4 framed** – The usual framing implementation of LZ4. For more information, see the [LZ4 frame format description](https://github.com/lz4/lz4) on GitHub.
  - **LZ4 hadoop-compatible** – The Apache Hadoop implementation of LZ4. This implementation wraps LZ4 compression with the `BlockCompressorStream.java` class.
- **LZO** – Format that uses the Lempel–Ziv–Oberhumer algorithm, which focuses on high compression and decompression speed rather than the maximum compression of data. LZO has two implementations:
  - **Standard LZO** – For more information, see the LZO abstract on the Oberhumer website.
  - **LZO hadoop-compatible** – This implementation wraps the LZO algorithm with the `BlockCompressorStream.java` class.
- **SNAPPY** – Compression algorithm that is part of the Lempel-Ziv 77 (LZ7) family. Snappy focuses on high compression and decompression speed rather than the maximum compression of data.

Some implementations of Snappy allow for framing. Framing enables decompression of streaming or file data that cannot be entirely maintained in memory. The following framing implementations are relevant for Athena:

- **Snappy Raw/Unframed** – The standard implementation of the Snappy format that does not use framing. For more information, see the [Snappy format description](https://github.com/google/snappy) on GitHub.
- **Snappy-framed** – The framing implementation of the Snappy format. For more information, see the [Snappy framing format description](https://github.com/google/snappy) on GitHub.
- **Snappy hadoop-compatible** – The framing implementation of Snappy that the Apache Hadoop Project uses. For more information, see `BlockCompressorStream.java` on GitHub.

For information about the Snappy framing methods that Athena supports for each file format, see the table later on this page.

- **ZLIB** – Based on Deflate. ZLIB is the default write compression format for files in the ORC data storage format. For more information, see the [zlib](https://github.com/gnu/zlib) page on GitHub.
- **ZSTD** – The Zstandard real-time data compression algorithm is a fast compression algorithm that provides high compression ratios. The Zstandard library is provided as open source software using a BSD license. Athena supports reading and writing ZStandard compressed ORC, Parquet, and text file data. When writing ZStandard compressed data, Athena uses ZStandard compression level 3.
Compression support in Athena by file format

Compression support in Athena depends on the engine version.

Compression support in Athena engine version 3

The following table summarizes the compression format support in Athena engine version 3 for each storage file format. Text file format includes TSV, CSV, JSON, and custom SerDes for text. "Yes" or "No" in a cell apply equally to read and write operations except where noted. For the purposes of this table, CREATE TABLE, CTAS, and INSERT INTO are considered write operations.

<table>
<thead>
<tr>
<th></th>
<th>Avro</th>
<th>Ion</th>
<th>ORC</th>
<th>Parquet</th>
<th>Text file</th>
</tr>
</thead>
<tbody>
<tr>
<td>BZIP2</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>DEFLATE</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>GZIP</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>LZ4</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>LZO</td>
<td>No</td>
<td>Write - No</td>
<td>No</td>
<td>Yes</td>
<td>Write - No Read - Yes</td>
</tr>
<tr>
<td>SNAPPY</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ZLIB</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ZSTD</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>NONE</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Compression support in Athena engine version 2

The following table summarizes the compression format support in Athena engine version 2. Text file format includes TSV, CSV, JSON, and custom SerDes for text. "Yes" or "No" in a cell apply equally to read and write operations except where noted. For the purposes of this table, CREATE TABLE, CTAS, and INSERT INTO are considered write operations.

<table>
<thead>
<tr>
<th></th>
<th>Avro</th>
<th>Ion</th>
<th>ORC</th>
<th>Parquet</th>
<th>Text file</th>
</tr>
</thead>
<tbody>
<tr>
<td>BZIP2</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>DEFLATE</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>GZIP</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>LZ4</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Write - Yes</td>
<td>Write - No Read - No</td>
</tr>
<tr>
<td>LZO</td>
<td>No</td>
<td>Write - No</td>
<td>No</td>
<td>Yes</td>
<td>Write - No Read - Yes</td>
</tr>
</tbody>
</table>
Specifying compression formats

When you write CREATE TABLE or CTAS statements, you can specify compression properties that specify the compression type to use when Athena writes to those tables.

- For CTAS, see CTAS table properties (p. 745). For examples, see Examples of CTAS queries (p. 286).
- For CREATE TABLE, see ALTER TABLE SET TBLPROPERTIES (p. 738) for a list of compression table properties.

Specifying no compression

CREATE TABLE statements support writing uncompressed files. To write uncompressed files, use the following syntax:

- CREATE TABLE (text file or JSON) – In TBLPROPERTIES, specify write.compression = NONE.
- CREATE TABLE (Parquet) – In TBLPROPERTIES, specify parquet.compression = UNCOMPRESSED.
- CREATE TABLE (ORC) – In TBLPROPERTIES, specify orc.compress = NONE.

Notes and resources

- Currently, uppercase file extensions such as .GZ or .BZIP2 are not recognized by Athena. Avoid using datasets with uppercase file extensions, or rename the data file extensions to lowercase.
- For data in CSV, TSV, and JSON, Athena determines the compression type from the file extension. If no file extension is present, Athena treats the data as uncompressed plain text. If your data is compressed, make sure the file name includes the compression extension, such as gz.
- The ZIP file format is not supported.
- For querying Amazon Kinesis Data Firehose logs from Athena, supported formats include GZIP compression or ORC files with SNAPPY compression.
- For more information about using compression, see section 3 ("Compress and split files") of the AWS Big Data Blog post Top 10 performance tuning tips for Amazon Athena.
SerDe reference

Athena supports several SerDe libraries for parsing data from different data formats, such as CSV, JSON, Parquet, and ORC. Athena does not support custom SerDes.

Topics

• Using a SerDe (p. 302)
• Supported SerDes and data formats (p. 303)

Using a SerDe

A SerDe (Serializer/Deserializer) is a way in which Athena interacts with data in various formats.

It is the SerDe you specify, and not the DDL, that defines the table schema. In other words, the SerDe can override the DDL configuration that you specify in Athena when you create your table.

To use a SerDe in queries

To use a SerDe when creating a table in Athena, use one of the following methods:

• Specify ROW FORMAT DELIMITED and then use DDL statements to specify field delimiters, as in the following example. When you specify ROW FORMAT DELIMITED, Athena uses the LazySimpleSerDe by default.

```
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
ESCAPED BY '\'
COLLECTION ITEMS TERMINATED BY '|' 
MAP KEYS TERMINATED BY ':'
```

For examples of ROW FORMAT DELIMITED, see the following topics:

LazySimpleSerDe for CSV, TSV, and custom-delimited files (p. 324)
Querying Amazon CloudFront logs (p. 452)
Querying Amazon EMR logs (p. 462)
Querying Amazon VPC flow logs (p. 474)
Using CTAS and INSERT INTO for ETL and data analysis (p. 290)

• Use ROW FORMAT SERDE to explicitly specify the type of SerDe that Athena should use when it reads and writes data to the table. The following example specifies the LazySimpleSerDe. To specify the delimiters, use WITH SERDEPROPERTIES. The properties specified by WITH SERDEPROPERTIES correspond to the separate statements (like FIELDS TERMINATED BY) in the ROW FORMAT DELIMITED example.

```
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'
```
### Supported SerDes and data formats

Athena supports creating tables and querying data from CSV, TSV, custom-delimited, and JSON formats; data from Hadoop-related formats: ORC, Apache Avro and Parquet; logs from Logstash, AWS CloudTrail logs, and Apache WebServer logs.

**Note**

The formats listed in this section are used by Athena for reading data. For information about formats that Athena uses for writing data when it runs CTAS queries, see Creating a table from query results (CTAS) (p. 281).

To create tables and query data in these formats in Athena, specify a serializer-deserializer class (SerDe) so that Athena knows which format is used and how to parse the data.

This table lists the data formats supported in Athena and their corresponding SerDe libraries.

A SerDe is a custom library that tells the data catalog used by Athena how to handle the data. You specify a SerDe type by listing it explicitly in the `ROW FORMAT` part of your `CREATE TABLE` statement in Athena. In some cases, you can omit the SerDe name because Athena uses some SerDe types by default for certain types of data formats.

#### Supported data formats and SerDes

<table>
<thead>
<tr>
<th>Data format</th>
<th>Description</th>
<th>SerDe types supported in Athena</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon Ion</td>
<td>Amazon Ion is a richly-typed, self-describing data format that is a superset of JSON, developed and open-sourced by Amazon.</td>
<td>Use the [Amazon Ion Hive SerDe](p. 305).</td>
</tr>
<tr>
<td>Apache Avro</td>
<td>A format for storing data in Hadoop that uses JSON-based schemas for record values.</td>
<td>Use the [Avro SerDe](p. 315).</td>
</tr>
<tr>
<td>Apache Parquet</td>
<td>A format for columnar storage of data in Hadoop.</td>
<td>Use the [Parquet SerDe](p. 334) and SNAPPY compression.</td>
</tr>
<tr>
<td>Data format</td>
<td>Description</td>
<td>SerDe types supported in Athena</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Apache WebServer logs</td>
<td>A format for storing logs in Apache WebServer.</td>
<td>Use the Grok SerDe (p. 317) or Regex SerDe (p. 337).</td>
</tr>
<tr>
<td>CloudTrail logs</td>
<td>A format for storing logs in CloudTrail.</td>
<td>Use the Hive JSON SerDe (p. 320). For more information, see Querying AWS CloudTrail logs (p. 454).</td>
</tr>
<tr>
<td>CSV (Comma-Separated Values)</td>
<td>For data in CSV, each line represents a data record, and each record consists of one or more fields, separated by commas.</td>
<td>Use the LazySimpleSerDe for CSV, TSV, and custom-delimited files (p. 324) if your data does not include values enclosed in quotes or if it uses the java.sql.Timestamp format. Use the OpenCSVSerDe for processing CSV (p. 329) when your data includes quotes in values or uses the UNIX numeric format for TIMESTAMP (for example, 1564610311).</td>
</tr>
<tr>
<td>Custom-Delimited</td>
<td>For data in this format, each line represents a data record, and records are separated by a custom single-character delimiter.</td>
<td>Use the LazySimpleSerDe for CSV, TSV, and custom-delimited files (p. 324) and specify a custom single-character delimiter.</td>
</tr>
<tr>
<td>JSON (JavaScript Object Notation)</td>
<td>For JSON data, each line represents a data record, and each record consists of attribute-value pairs and arrays, separated by commas.</td>
<td>Use the Hive JSON SerDe (p. 320). Use the OpenX JSON SerDe (p. 321).</td>
</tr>
<tr>
<td>Logstash logs</td>
<td>A format for storing logs in Logstash.</td>
<td>Use the Grok SerDe (p. 317).</td>
</tr>
<tr>
<td>ORC (Optimized Row Columnar)</td>
<td>A format for optimized columnar storage of Hive data.</td>
<td>Use the ORC SerDe (p. 332) and ZLIB compression.</td>
</tr>
<tr>
<td>TSV (Tab-Separated Values)</td>
<td>For data in TSV, each line represents a data record, and each record consists of one or more fields, separated by tabs.</td>
<td>Use the LazySimpleSerDe for CSV, TSV, and custom-delimited files (p. 324) and specify the separator character as FIELDS TERMINATED BY ‘\t’.</td>
</tr>
</tbody>
</table>

**Topics**
- Amazon Ion Hive SerDe (p. 305)
- Avro SerDe (p. 315)
- Grok SerDe (p. 317)
- JSON SerDe libraries (p. 319)
• LazySimpleSerDe for CSV, TSV, and custom-delimited files (p. 324)
• OpenCSVSerDe for processing CSV (p. 329)
• ORC SerDe (p. 332)
• Parquet SerDe (p. 334)
• Regex SerDe (p. 337)

Amazon Ion Hive SerDe

Amazon Ion has binary and text formats that are interchangeable. This feature combines the ease of use of text with the efficiency of binary encoding.

To query Amazon Ion data from Athena, you can use the Amazon Ion Hive SerDe, which serializes and deserializes Amazon Ion data. Deserialization allows you to run queries on the Amazon Ion data or read it for writing out into a different format like Parquet or ORC. Serialization lets you generate data in the Amazon Ion format by using CREATE TABLE AS SELECT (CTAS) or INSERT INTO queries to copy data from existing tables.

**Note**
Because Amazon Ion is a superset of JSON, you can use the Amazon Ion Hive SerDe to query non-Amazon Ion JSON datasets. Unlike other JSON SerDe libraries, the Amazon Ion SerDe does not expect each row of data to be on a single line. This feature is useful if you want to query JSON datasets that are in "pretty print" format or otherwise break up the fields in a row with newline characters.

**SerDe name**

• com.amazon.ionhiveserde.IonHiveSerDe

**Considerations and limitations**

• **Duplicated fields** – Amazon Ion structs are ordered and support duplicated fields, while Hive's STRUCT<> and MAP<> do not. Thus, when you deserialize a duplicated field from an Amazon Ion struct, a single value is chosen non deterministically, and the others are ignored.

• **External symbol tables unsupported** – Currently, Athena does not support external symbol tables or the following Amazon Ion Hive SerDe properties:
  • ion.catalog.class
  • ion.catalog.file
  • ion.catalog.url
  • ion.symbol_table_imports

• **File extensions** – Amazon Ion uses file extensions to determine which compression codec to use for deserializing Amazon Ion files. As such, compressed files must have the file extension that corresponds to the compression algorithm used. For example, if ZSTD is used, corresponding files should have the extension .zst.

• **Homogeneous data** – Amazon Ion has no restrictions on the data types that can be used for values in particular fields. For example, two different Amazon Ion documents might have a field with the same name that have different data types. However, because Hive uses a schema, all values that you extract to a single Hive column must have the same data type.
• **Map key type restrictions** – When you serialize data from another format into Amazon Ion, ensure that the map key type is one of STRING, VARCHAR, or CHAR. Although Hive allows you to use any primitive data type as a map key, Amazon Ion symbols must be a string type.

• **Union type** – Athena does not currently support the Hive union type.

Topics

• Using CREATE TABLE to create Amazon Ion tables (p. 306)
• Using CTAS and INSERT INTO to create Amazon Ion tables (p. 307)
• Using Amazon Ion SerDe properties (p. 308)
• Using path extractors (p. 312)

Using CREATE TABLE to create Amazon Ion tables

To create a table in Athena from data stored in Amazon Ion format, you can use one of the following techniques in a CREATE TABLE statement:

• Specify STORED AS ION. In this usage, you do not have to specify the Amazon Ion Hive SerDe explicitly. This choice is the more straightforward option.

• Specify the Amazon Ion class paths in the \texttt{ROW FORMAT SERDE}, \texttt{INPUTFORMAT}, and \texttt{OUTPUTFORMAT} fields.

You can also use \texttt{CREATE TABLE AS SELECT (CTAS)} statements to create Amazon Ion tables in Athena. For information, see Using CTAS and INSERT INTO to create Amazon Ion tables (p. 307).

Specifying STORED AS ION

The following example \texttt{CREATE TABLE} statement uses \texttt{STORED AS ION} before the \texttt{LOCATION} clause to create a table based on flight data in Amazon Ion format. The \texttt{LOCATION} clause specifies the bucket or folder where the input files in Ion format are located. All files in the specified location are scanned.

```sql
CREATE EXTERNAL TABLE flights_ion (
    yr INT,
    quarter INT,
    month INT,
    dayofmonth INT,
    dayofweek INT,
    flightdate STRING,
    uniquecarrier STRING,
    airlineid INT,
)
STORED AS ION
LOCATION 's3://DOC-EXAMPLE-BUCKET/
```

Specifying the Amazon Ion class paths

Instead of using the \texttt{STORED AS ION} syntax, you can explicitly specify the Ion class path values for the \texttt{ROW FORMAT SERDE}, \texttt{INPUTFORMAT}, and \texttt{OUTPUTFORMAT} clauses as follows.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Ion class path</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROW FORMAT SERDE</td>
<td>'com.amazon.ionhiveserde.IonHiveSerDe'</td>
</tr>
</tbody>
</table>
Parameter | Ion class path
---|---
STORED AS INPUTFORMAT | 'com.amazon.ionhiveserde.formats.IonInputFormat'
OUTPUTFORMAT | 'com.amazon.ionhiveserde.formats.IonOutputFormat'

The following DDL query uses this technique to create the same external table as in the previous example.

```sql
CREATE EXTERNAL TABLE flights_ion (
    yr INT,
    quarter INT,
    month INT,
    dayofmonth INT,
    dayofweek INT,
    flightdate STRING,
    uniquecarrier STRING,
    airlineid INT,
) ROW FORMAT SERDE
    'com.amazon.ionhiveserde.IonHiveSerDe'
STORED AS INPUTFORMAT
    'com.amazon.ionhiveserde.formats.IonInputFormat'
OUTPUTFORMAT
    'com.amazon.ionhiveserde.formats.IonOutputFormat'
LOCATION 's3://DOC-EXAMPLE-BUCKET/
```

For information about the SerDe properties for CREATE TABLE statements in Athena, see Using Amazon Ion SerDe properties (p. 308).

**Using CTAS and INSERT INTO to create Amazon Ion tables**

You can use the CREATE TABLE AS SELECT (CTAS) and INSERT INTO statements to copy or insert data from a table into a new table in Amazon Ion format in Athena.

In a CTAS query, specify format='ION' in the WITH clause, as in the following example.

```sql
CREATE TABLE new_table
WITH (format='ION')
AS SELECT * from existing_table
```

By default, Athena serializes Amazon Ion results in Ion binary format, but you can also use text format. To use text format, specify ion_encoding = 'TEXT' in the CTAS WITH clause, as in the following example.

```sql
CREATE TABLE new_table
WITH (format='ION', ion_encoding = 'TEXT')
AS SELECT * from existing_table
```

For more information about Amazon Ion specific properties in the CTAS WITH clause, see the following section.

**CTAS WITH clause Amazon Ion properties**

In a CTAS query, you can use the WITH clause to specify the Amazon Ion format and optionally specify the Amazon Ion encoding and/or write compression algorithm to use.
format

You can specify the ION keyword as the format option in the WITH clause of a CTAS query. When you do so, the table that you create uses the format that you specify for IonInputFormat for reads, and it serializes data in the format that you specify for IonOutputFormat.

The following example specifies that the CTAS query use Amazon Ion format.

```
WITH (format='ION')
```

ion_encoding

Optional

Default: BINARY

Values: BINARY, TEXT

Specifies whether data is serialized in Amazon Ion binary format or Amazon Ion text format. The following example specifies Amazon Ion text format.

```
WITH (format='ION', ion_encoding='TEXT')
```

write_compression

Optional

Default: GZIP

Values: GZIP, ZSTD, BZIP2, SNAPPY, NONE

Specifies the compression algorithm to use to compress output files.

The following example specifies that the CTAS query write its output in Amazon Ion format using the Zstandard compression algorithm.

```
WITH (format='ION', write_compression = 'ZSTD')
```

For information about using compression in Athena, see Athena compression support (p. 299).

For information about other CTAS properties in Athena, see CTAS table properties (p. 745).

Using Amazon Ion SerDe properties

This topic contains information about the SerDe properties for CREATE TABLE statements in Athena. For more information and examples of Amazon Ion SerDe property usage, see SerDe properties in the Amazon Ion Hive SerDe documentation on GitHub.

Specifying Amazon Ion SerDe properties

To specify properties for the Amazon Ion Hive SerDe in your CREATE TABLE statement, use the WITH SERDEPROPERTIES clause. Because WITH SERDEPROPERTIES is a subfield of the ROW FORMAT SERDE clause, you must specify ROW FORMAT SERDE and the Amazon Ion Hive SerDe class path first, as the following syntax shows.

```
...
ROW FORMAT SERDE
'com.amazon.ionhiveserde.IonHiveSerDe'
WITH SERDEPROPERTIES (
  'property' = 'value',
  'property' = 'value',
  ...
)

Note that although the ROW FORMAT SERDE clause is required if you want to use WITH SERDEPROPERTIES, you can use either STORED AS ION or the longer INPUTFORMAT and OUTPUTFORMAT syntax to specify the Amazon Ion format.

Amazon Ion SerDe properties

Following are the Amazon Ion SerDe properties that can be used in CREATE TABLE statements in Athena.

ion.encoding

Optional

Default: BINARY

Values: BINARY, TEXT

This property declares whether new values added are serialized as Amazon Ion binary or Amazon Ion text format.

The following SerDe property example specifies Amazon Ion text format.

'ion.encoding' = 'TEXT'

ion.fail_on_overflow

Optional

Default: true

Values: true, false

Amazon Ion allows for arbitrarily large numerical types while Hive does not. By default, the SerDe fails if the Amazon Ion value does not fit the Hive column, but you can use the fail_on_overflow configuration option to let the value overflow instead of failing.

This property can be set at either the table or column level. To specify it at the table level, specify ion.fail_on_overflow as in the following example. This sets the default behavior for all columns.

'ion.fail_on_overflow' = 'true'

To control a specific column, specify the column name between ion and fail_on_overflow, delimited by periods, as in the following example.

'ion.<column>.fail_on_overflow' = 'false'

ion.path_extractor.case_sensitive

Optional
Default: false

Values: true, false

Determines whether to treat Amazon Ion field names as case sensitive. When false, the SerDe ignores case parsing Amazon Ion field names.

For example, suppose you have a Hive table schema that defines a field alias in lower case and an Amazon Ion document with both an alias field and an ALIAS field, as in the following example.

```
-- Hive Table Schema
alias: STRING

-- Amazon Ion Document
{ 'ALIAS': 'value1'
{ 'alias': 'value2'
```

The following example shows SerDe properties and the resulting extracted table when case sensitivity is set to false:

```
-- Serde properties
'ion.alias.path_extractor' = '(alias)' 
'ion.path_extractor.case_sensitive' = 'false'

--Extracted Table
<table>
<thead>
<tr>
<th>alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;value1&quot;</td>
</tr>
<tr>
<td>&quot;value2&quot;</td>
</tr>
</tbody>
</table>
```

The following example shows SerDe properties and the resulting extracted table when case sensitivity is set to true:

```
-- Serde properties
'ion.alias.path_extractor' = '(alias)' 
'ion.path_extractor.case_sensitive' = 'true'

--Extracted Table
<table>
<thead>
<tr>
<th>alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;value2&quot;</td>
</tr>
</tbody>
</table>
```

In the second case, value1 for the ALIAS field is ignored when case sensitivity is set to true and the path extractor is specified as alias.

**ion.<column>.path_extractor**

Optional

Default: NA

Values: String with search path

Creates a path extractor with the specified search path for the given column. Path extractors map Amazon Ion fields to Hive columns. If no path extractors are specified, Athena dynamically creates path extractors at run time based on column names.

The following example path extractor maps the example_ion_field to the example_hive_column.
'ion.example_hive_column.path_extractor' = '(example_ion_field)'

For more information about path extractors and search paths, see Using path extractors (p. 312).

**ion.timestamp.serialization_offset**

Optional

Default: 'Z'

Values: OFFSET, where OFFSET is represented as <signal>hh:mm. Example values: 01:00, +01:00, -09:30, Z (UTC, same as 00:00)

Unlike Apache Hive timestamps, which have no built-in time zone and are stored as an offset from the UNIX epoch, Amazon Ion timestamps do have an offset. Use this property to specify the offset when you serialize to Amazon Ion.

The following example adds an offset of one hour.

'ion.timestamp.serialization_offset' = '+01:00'

**ion.serialize_null**

Optional

Default: OMIT

Values: OMIT, UNTYPED, TYPED

The Amazon Ion SerDe can be configured to either serialize or omit columns that have null values. You can choose to write out strongly typed nulls (TYPED) or untyped nulls (UNTYPED). Strongly typed nulls are determined based on the default Amazon Ion to Hive type mapping.

The following example specifies strongly typed nulls.

'ion.serialize_null'='TYPED'

**ion.ignore_malformed**

Optional

Default: false

Values: true, false

When true, ignores malformed entries or the whole file if the SerDe is unable to read it. For more information, see Ignore malformed in the documentation on GitHub.

**ion.<column>.serialize_as**

Optional

Default: Default type for the column.

Values: String containing Amazon Ion type

Determines the Amazon Ion data type in which a value is serialized. Because Amazon Ion and Hive types do not always have a direct mapping, a few Hive types have multiple valid data types for
serialization. To serialize data as a non-default data type, use this property. For more information about type mapping, see the Amazon Ion Type mapping page on GitHub.

By default, binary Hive columns are serialized as Amazon Ion blobs, but they can also be serialized as an Amazon Ion clob (character large object). The following example serializes the column example_hive_binary_column as a clob.

'ion.example_hive_binary_column.serialize_as' = 'clob'

Using path extractors

Amazon Ion is a document style file format, but Apache Hive is a flat columnar format. You can use special Amazon Ion SerDe properties called path extractors to map between the two formats. Path extractors flatten the hierarchical Amazon Ion format, map Amazon Ion values to Hive columns, and can be used to rename fields.

Athena can generate the extractors for you, but you can also define your own extractors if necessary.

Generated path extractors

By default, Athena searches for top level Amazon Ion values that match Hive column names and creates path extractors at runtime based on these matching values. If your Amazon Ion data format matches the Hive table schema, Athena dynamically generates the extractors for you, and you do not need to add any additional path extractors. These default path extractors are not stored in the table metadata.

The following example shows how Athena generates extractors based on column name.

-- Example Amazon Ion Document
{
  identification: {
    name: "John Smith",
    driver_license: "XXXX"
  },
  alias: "Johnny"
}

-- Example DDL
CREATE EXTERNAL TABLE example_schema2 (identification MAP<STRING, STRING>, alias STRING)
STORED AS ION
LOCATION 's3://DOC-EXAMPLE-BUCKET/path_extraction1'/

The following example extractors are generated by Athena. The first extracts the identification field to the identification column, and the second extracts the alias field to the alias column.

'ion.identification.path_extractor' = '(identification)'
'ion.alias.path_extractor' = '(alias)'

The following example shows the extracted table.

<table>
<thead>
<tr>
<th>identification</th>
<th>alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;[&quot;name&quot;, &quot;driver_license&quot;],[&quot;John Smith&quot;, &quot;XXXX&quot;]&quot;</td>
<td>&quot;Johnny&quot;</td>
</tr>
</tbody>
</table>
Specifying your own path extractors

If your Amazon Ion fields do not map neatly to Hive columns, you can specify your own path extractors. In the WITH SERDEPROPERTIES clause of your CREATE TABLE statement, use the following syntax.

```sql
WITH SERDEPROPERTIES (  
    "ion.path_extractor.case_sensitive" = "<Boolean>",  
    "ion.<column_name>.path_extractor" = "<path_extractor_expression>"
)
```

**Note**
By default, path extractors are case insensitive. To override this setting, set the `ion.path_extractor.case_sensitive` SerDe property to `true`.

Using search paths in path extractors

The SerDe property syntax for path extractor contains a `<path_extractor_expression>`:

```sql
"ion.<column_name>.path_extractor" = "<path_extractor_expression>"
```

You can use the `<path_extractor_expression>` to specify a search path that parses the Amazon Ion document and finds matching data. The search path is enclosed in parenthesis and can contain one or more of the following components separated by spaces.

- **Wild card** – Matches all values.
- **Index** – Matches the value at the specified numerical index. Indices are zero-based.
- **Text** – Matches all values whose field names match are equivalent to the specified text.
- **Annotations** – Matches values specified by a wrapped path component that has the annotations specified.

The following example shows an Amazon Ion document and some example search paths.

```json
-- Amazon Ion document
{
    foo: ["foo1", "foo2"],
    bar: "myBarValue",
    bar: A::"annotatedValue"
}

-- Example search paths
(foo 0) # matches "foo1"
(1) # matches "myBarValue"
(*) # matches ["foo1", "foo2"], "myBarValue" and A::"annotatedValue"
() # matches {foo: ["foo1", "foo2"], bar: "myBarValue", bar: A::"annotatedValue"}
(bar) # matches "myBarValue" and A::"annotatedValue"
(A::bar) # matches A::"annotatedValue"
```

Extractor examples

Flattening and renaming fields

The following example shows a set of search paths that flatten and rename fields. The example uses search paths to do the following:

- Map the nickname column to the alias field
- Map the name column to the name subfield located in the identification struct.

Following is the example Amazon Ion document.

```java
-- Example Amazon Ion Document
{
  identification: {
    name: "John Smith",
    driver_license: "XXXX"
  },
  alias: "Johnny"
}
```

The following is the example CREATE TABLE statement that defines the path extractors.

```sql
-- Example DDL Query
CREATE EXTERNAL TABLE example_schema2 (  
  name STRING,
  nickname STRING
)  
ROW FORMAT SERDE  
'com.amazon.ionhiveserde.IonHiveSerDe'  
WITH SERDEPROPERTIES (  
  'ion.nickname.path_extractor' = '(alias)',  
  'ion.name.path_extractor' = '(identification name)'  
)  
STORED AS ION  
LOCATION 's3://DOC-EXAMPLE-BUCKET/path_extraction2/'
```

The following example shows the extracted data.

```text
<table>
<thead>
<tr>
<th>name</th>
<th>nickname</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;John Smith&quot;</td>
<td>&quot;Johnny&quot;</td>
</tr>
</tbody>
</table>
```

For more information about search paths and additional search path examples, see the Ion Java Path Extraction page on GitHub.

**Extracting flight data to text format**

The following example CREATE TABLE query uses WITH SERDEPROPERTIES to add path extractors to extract flight data and specify the output encoding as Amazon Ion text. The example uses the STORED AS ION syntax.

```sql
CREATE EXTERNAL TABLE flights_ion (  
  yr INT,  
  quarter INT,  
  month INT,  
  dayofmonth INT,  
  dayofweek INT,  
  flightdate STRING,  
  uniquecarrier STRING,  
  airlineid INT,  
)  
ROW FORMAT SERDE  
'com.amazon.ionhiveserde.IonHiveSerDe'  
WITH SERDEPROPERTIES (  
  'ion.encoding' = 'TEXT',
```
Avro SerDe

SerDe name
Avro SerDe

Library name
org.apache.hadoop.hive.serde2.avro.AvroSerDe

Examples

Athena does not support using `avro.schema.url` to specify table schema for security reasons. Use `avro.schema.literal`. To extract schema from data in the Avro format, use the Apache `avro-tools-<version>.jar` with the `getschema` parameter. This returns a schema that you can use in your `WITH SERDEPROPERTIES` statement. For example:

```java
java -jar avro-tools-1.8.2.jar getschema my_data.avro
```

The `avro-tools-<version>.jar` file is located in the `java` subdirectory of your installed Avro release. To download Avro, see [Apache Avro releases](#). To download Apache Avro Tools directly, see the [Apache Avro tools Maven repository](#).

After you obtain the schema, use a `CREATE TABLE` statement to create an Athena table based on underlying Avro data stored in Amazon S3. To specify the Avro SerDe, use `ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'`. As demonstrated in the following example, you must specify the schema using the `WITH SERDEPROPERTIES` clause in addition to specifying the column names and corresponding data types for the table.

**Note**
Replace `myregion` in `s3://athena-examples-myregion/path/to/data/` with the region identifier where you run Athena, for example, `s3://athena-examples-us-west-1/path/to/data/`.

```sql
CREATE EXTERNAL TABLE flights_avro_example (yr INT,
    flightdate STRING,
    uniquecarrier STRING,
    airlineid INT,
    carrier STRING,
    flightnum STRING,
    origin STRING,
    dest STRING,
    depdelay INT,
    carrierdelay INT,
    weatherdelay INT)
PARTITIONED BY (year STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
WITH SERDEPROPERTIES ('avro.schema.literal'=
{
  "type" : "record",
  "name" : "flights_avro_subset",
  "fields": ["yr", "flightdate", "uniquecarrier", "airlineid", "carrier", "flightnum", "origin", "dest", "depdelay", "carrierdelay", "weatherdelay"]
})
```

STORED AS ION
LOCATION 's3://DOC-EXAMPLE-BUCKET/'
"namespace" : "default",
"fields" : [ {
  "name" : "yr",
  "type" : [ "null", "int" ],
  "default" : null
}, {
  "name" : "flightdate",
  "type" : [ "null", "string" ],
  "default" : null
}, {
  "name" : "uniquecarrier",
  "type" : [ "null", "string" ],
  "default" : null
}, {
  "name" : "airlineid",
  "type" : [ "null", "int" ],
  "default" : null
}, {
  "name" : "carrier",
  "type" : [ "null", "string" ],
  "default" : null
}, {
  "name" : "flightnum",
  "type" : [ "null", "string" ],
  "default" : null
}, {
  "name" : "origin",
  "type" : [ "null", "string" ],
  "default" : null
}, {
  "name" : "dest",
  "type" : [ "null", "string" ],
  "default" : null
}, {
  "name" : "depdelay",
  "type" : [ "null", "int" ],
  "default" : null
}, {
  "name" : "carrierdelay",
  "type" : [ "null", "int" ],
  "default" : null
}, {
  "name" : "weatherdelay",
  "type" : [ "null", "int" ],
  "default" : null
} ]
}

STORED AS AVRO
LOCATION 's3://athena-examples-myregion/flight/avro/';

Run the **MSCK REPAIR TABLE** statement on the table to refresh partition metadata.

Run the **MSCK REPAIR TABLE** flights_avro_example;

Query the top 10 departure cities by number of total departures.

```
SELECT origin, count(*) AS total_departures
FROM flights_avro_example
WHERE year >= '2000'
GROUP BY origin
ORDER BY total_departures DESC
LIMIT 10;
```
**Grok SerDe**

The Logstash Grok SerDe is a library with a set of specialized patterns for deserialization of unstructured text data, usually logs. Each Grok pattern is a named regular expression. You can identify and re-use these deserialization patterns as needed. This makes it easier to use Grok compared with using regular expressions. Grok provides a set of pre-defined patterns. You can also create custom patterns.

To specify the Grok SerDe when creating a table in Athena, use the `ROW FORMAT SERDE 'com.amazonaws.glue.serde.GrokSerDe'` clause, followed by the `WITH SERDEPROPERTIES` clause that specifies the patterns to match in your data, where:

- The `input.format` expression defines the patterns to match in the data. It is required.
- The `input.grokCustomPatterns` expression defines a named custom pattern, which you can subsequently use within the `input.format` expression. It is optional. To include multiple pattern entries into the `input.grokCustomPatterns` expression, use the newline escape character (`\n`) to separate them, as follows: `input.grokCustomPatterns='INSIDE_QS ([\"\]*|\n)INSIDE_BRACKETS ([^\[]\])

- The `STORED AS INPUTFORMAT` and `OUTPUTFORMAT` clauses are required.
- The `LOCATION` clause specifies an Amazon S3 bucket, which can contain multiple data objects. All data objects in the bucket are deserialized to create the table.

**Examples**

These examples rely on the list of predefined Grok patterns. See pre-defined patterns.

**Example 1**

This example uses source data from Postfix maillog entries saved in `s3://mybucket/groksample/`.

```
Feb 9 07:15:00 m4eastmail postfix/smtpd[19305]: B88C4120838: connect from unknown[192.168.55.4]
Feb 9 07:15:00 m4eastmail postfix/smtpd[20444]: B58C4330038: client=unknown[192.168.55.4]
Feb 9 07:15:03 m4eastmail postfix/cleanup[22835]: BDC22A77854: message-id=<31221401257553.5004389LCBF@m4eastmail.example.com>
```

The following statement creates a table in Athena called `mygroktable` from the source data, using a custom pattern and the predefined patterns that you specify:

```
CREATE EXTERNAL TABLE `mygroktable`(
    syslogbase string,
    queue_id string,
    syslog_message string
)
ROW FORMAT SERDE 'com.amazonaws.glue.serde.GrokSerDe'
WITH SERDEPROPERTIES (  'input.grokCustomPatterns' = 'POSTFIX_QUEUEID [0-9A-F]{7,12}',
    'input.format'='%(SYSLOGBASE) %(POSTFIX_QUEUEID:queue_id): %(GREEDYDATA:syslog_message)'  )
STORED AS INPUTFORMAT 'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
```
LOCATION
's3://mybucket/groksample/';

Start with a simple pattern, such as %{NOTSPACE:column}, to get the columns mapped first and then specialize the columns if needed.

Example 2

In the following example, you create a query for Log4j logs. The example logs have the entries in this format:

```
2017-09-12 12:10:34,972 INFO  - processType=AZ, processId=ABCDEFG614B6F5E49, status=RUN, threadId=123:amqListenerContainerPool123:P:AJ|ABCDEFG614B6F5E49]| executionTime=7290, tenantId=12456, userId=123123f8535f8d76015374e7a1d87c3c, shard=testapp1, jobId=12312345e7d0015e777fb2e03f3c, messageType=REAL_TIME_SYNC, action=receive, hostname=1.abc.def.com
```

To query this logs data:

- Add the Grok pattern to the `input.format` for each column. For example, for timestamp, add `%{TIMESTAMP_ISO8601:timestamp}`. For loglevel, add `%{LOGLEVEL:loglevel}`.
- Make sure the pattern in `input.format` matches the format of the log exactly, by mapping the dashes (-) and the commas that separate the entries in the log format.

```sql
CREATE EXTERNAL TABLE bltest (
    timestamp STRING,
    loglevel STRING,
    processType STRING,
    processId STRING,
    status STRING,
    threadId STRING,
    executionTime INT,
    tenantId INT,
    userId STRING,
    shard STRING,
    jobId STRING,
    messageType STRING,
    action STRING,
    hostname STRING
) ROW FORMAT SERDE 'com.amazonaws.glue.serde.GrokSerDe'
WITH SERDEPROPERTIES ("input.grokCustomPatterns" = 'C_ACTION receive|send', "input.format" = "%{TIMESTAMP_ISO8601:timestamp} %{LOGLEVEL:loglevel} - processType=%%(NOTSPACE:processType) %{NOTSPACE:processId}, status=%%(NOTSPACE:status), threadId=%%(NOTSPACE:threadId), executionTime=%%(POSINT:executionTime), tenantId=%%(POSINT:tenantId), userId=%%(NOTSPACE:userId), shard=%%(NOTSPACE:shard), jobId=%%(NOTSPACE:jobId), messageType=%%(NOTSPACE:messageType), action=%%(C_ACTION:action), hostname=%%(HOST:hostname)"
```

Example 3

The following example of querying Amazon S3 logs shows the `input.grokCustomPatterns` expression that contains two pattern entries, separated by the newline escape character (\n), as
CREATE EXTERNAL TABLE `s3_access_auto_raw_02`(
  `bucket_owner` string COMMENT 'from deserializer',
  `bucket` string COMMENT 'from deserializer',
  `time` string COMMENT 'from deserializer',
  `remote_ip` string COMMENT 'from deserializer',
  `requester` string COMMENT 'from deserializer',
  `request_id` string COMMENT 'from deserializer',
  `operation` string COMMENT 'from deserializer',
  `key` string COMMENT 'from deserializer',
  `request_uri` string COMMENT 'from deserializer',
  `http_status` string COMMENT 'from deserializer',
  `error_code` string COMMENT 'from deserializer',
  `bytes_sent` string COMMENT 'from deserializer',
  `object_size` string COMMENT 'from deserializer',
  `total_time` string COMMENT 'from deserializer',
  `turnaround_time` string COMMENT 'from deserializer',
  `referrer` string COMMENT 'from deserializer',
  `user_agent` string COMMENT 'from deserializer',
  `version_id` string COMMENT 'from deserializer') ROW FORMAT SERDE
  'com.amazonaws.glue.serde.GrokSerDe'
  WITH SERDEPROPERTIES (
    'input.format'='%{NOTSPACE:bucket_owner} %{NOTSPACE:bucket} \[%{INSIDE_BRACKETS:time}\]%
    %{NOTSPACE:remote_ip} %{NOTSPACE:requester} %{NOTSPACE:request_id} %{NOTSPACE:operation}
    %{NOTSPACE:key} "?%{INSIDE_QS:request_uri}"? %{NOTSPACE:http_status}
    %{NOTSPACE:error_code} %{NOTSPACE:bytes_sent} %{NOTSPACE:object_size}
    %{NOTSPACE:total_time} %{NOTSPACE:turnaround_time} "?%{INSIDE_QS:referrer}"? "?
    %{INSIDE_QS:referrer}\"? %{NOTSPACE:version_id}',
    'input.grokCustomPatterns'='INSIDE_QS ([^"\"]*)
    INSIDE_BRACKETS ([^\\]*)') STORED AS INPUTFORMAT
  'org.apache.hadoop.mapred.TextInputFormat'
  OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
  LOCATION
  's3://bucket-for-service-logs/s3_access/'

JSON SerDe libraries

In Athena, you can use SerDe libraries to deserialize JSON data. Deserialization converts the JSON data so that it can be serialized (written out) into a different format like Parquet or ORC.

- The native Hive JSON SerDe (p. 320)
- The OpenX JSON SerDe (p. 321)
- The Amazon Ion Hive SerDe (p. 305)

**Note**

The Hive and OpenX libraries expect JSON data to be on a single line (not formatted), with records separated by a new line character. The Amazon Ion Hive SerDe does not have that requirement and can be used as an alternative because the Ion data format is a superset of JSON.

**Library names**

Use one of the following:

```
org.apache.hive.hcatalog.data.JsonSerDe```
Hive JSON SerDe

The Hive JSON SerDe is commonly used to process JSON data like events. These events are represented as single-line strings of JSON-encoded text separated by a new line. The Hive JSON SerDe does not allow duplicate keys in map or struct key names.

Note
The SerDe expects each JSON document to be on a single line of text with no line termination characters separating the fields in the record. If the JSON text is in pretty print format, you may receive an error message like HIVE_CURSOR_ERROR: Row is not a valid JSON Object or HIVE_CURSOR_ERROR: JsonParseException: Unexpected end-of-input: expected close marker for OBJECT when you attempt to query the table after you create it. For more information, see JSON Data Files in the OpenX SerDe documentation on GitHub.

The following example DDL statement uses the Hive JSON SerDe to create a table based on sample online advertising data. In the LOCATION clause, replace the myregion in s3://myregion.elasticmapreduce/samples/hive-ads/tables/impressions with the region identifier where you run Athena (for example, s3://us-west-2.elasticmapreduce/samples/hive-ads/tables/impressions).

```sql
CREATE EXTERNAL TABLE impressions (  
  requestbegintime string,  
  adid string,  
  impressionid string,  
  referrer string,  
  useragent string,  
  usercookie string,  
  ip string,  
  number string,  
  processid string,  
  browsercookie string,  
  requestendtime string,  
  timers struct  
  <  
    modellookup:string,  
    requesttime:string  
  >,  
  threadid string,  
  hostname string,  
  sessionid string  
) PARTITIONED BY (dt string)  
ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe'  
LOCATION 's3://myregion.elasticmapreduce/samples/hive-ads/tables/impressions';
```

Specifying timestamp formats with the Hive JSON SerDe

To parse timestamp values from string, you can add the WITH SERDEPROPERTIES subfield to the ROW FORMAT SERDE clause and use it to specify the timestamp.formats parameter. In the parameter, specify a comma-separated list of one or more timestamp patterns, as in the following example:

```sql
...  
ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe'  
WITH SERDEPROPERTIES ('timestamp.formats'="yyyy-MM-dd'T'HH:mm:ss.SSS'Z',yyyy-MM-dd'T'HH:mm:ss")  
...
For more information, see Timestamps in the Apache Hive documentation.

**Loading the table for querying**

After you create the table, run `MSCK REPAIR TABLE (p. 756)` to load the table and make it queryable from Athena:

```
MSCK REPAIR TABLE impressions
```

**Querying CloudTrail logs**

You can use the Hive JSON SerDe to query CloudTrail logs. For more information and example `CREATE TABLE` statements, see Querying AWS CloudTrail logs (p. 454).

**OpenX JSON SerDe**

In addition to the `paths` property for the columns in the table, the OpenX JSON SerDe has the following optional properties that can be useful for addressing inconsistencies in data.

- **ignore.malformed.json**
  - Optional. When set to TRUE, lets you skip malformed JSON syntax. The default is FALSE.

- **dots.in.keys**
  - Optional. The default is FALSE. When set to TRUE, allows the SerDe to replace the dots in key names with underscores. For example, if the JSON dataset contains a key with the name "a.b", you can use this property to define the column name to be "a_b" in Athena. By default (without this SerDe), Athena does not allow dots in column names.

- **case.insensitive**
  - Optional. The default is TRUE. When set to TRUE, the SerDe converts all uppercase columns to lowercase.
  
  To use case-sensitive key names in your data, use `WITH SERDEPROPERTIES ("case.insensitive"= FALSE;)`. Then, for every key that is not already all lowercase, provide a mapping from the column name to the property name using the following syntax:

```
ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'
WITH SERDEPROPERTIES ("case.insensitive" = "FALSE", "mapping.userid" = "userId")
```

- **mapping**
  - Optional. Maps column names to JSON keys that aren't identical to the column names. The mapping parameter is useful when the JSON data contains keys that are keywords (p. 256). For example,
if you have a JSON key named timestamp, use the following syntax to map the key to a column named ts:

```
ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'
WITH SERDEPROPERTIES ("mapping.ts"= "timestamp")
```

Like the Hive JSON SerDe, the OpenX JSON SerDe does not allow duplicate keys in map or struct key names.

The following example DDL statement uses the OpenX JSON SerDe to create a table based on the same sample online advertising data used in the example for the Hive JSON SerDe. In the LOCATION clause, replace `myregion` with the region identifier where you run Athena.

```
CREATE EXTERNAL TABLE impressions (  
    requestbegintime string,  
    adid string,  
    impressionId string,  
    referrer string,  
    useragent string,  
    usercookie string,  
    ip string,  
    number string,  
    processid string,  
    browsercokie string,  
    requestendtime string,  
    timers struct<  
        modellookup:string,  
        requesttime:string>,  
    threadid string,  
    hostname string,  
    sessionid string  
) PARTITIONED BY (dt string)
ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'
with serdeproperties ( 'paths'='requestbegintime, adid, impressionid, referrer, useragent, usercookie, ip' )
LOCATION 's3://myregion.elasticmapreduce/samples/hive-ads/tables/impressions';
```

**Note**

The SerDe expects each JSON document to be on a single line of text with no line termination characters separating the fields in the record. If the JSON text is in pretty print format, you may receive an error message like HIVE_CURSOR_ERROR: Row is not a valid JSON Object or HIVE_CURSOR_ERROR: JsonParseException: Unexpected end-of-input: expected close marker for OBJECT when you attempt to query the table after you create it. For more information, see JSON Data Files in the OpenX SerDe documentation on GitHub.

**Example: Deserializing nested JSON**

You can use the JSON SerDes to parse more complex JSON-encoded data. This requires using CREATE TABLE statements that use struct and array elements to represent nested structures.

The following example creates an Athena table from JSON data that has nested structures. To parse JSON-encoded data in Athena, make sure that each JSON document is on its own line, separated by a new line.

This example presumes JSON-encoded data that has the following structure:

```json
{
    "DocId": "AWS",
```
The following CREATE TABLE statement uses the OpenX-JsonSerDe with the struct and array collection data types to establish groups of objects. Each JSON document is listed on its own line, separated by a new line. To avoid errors, the data being queried does not include duplicate keys in struct or map key names.

```
CREATE external TABLE complex_json (
    docid string,
    'user' struct<
        id:INT,
        username:string,
        name:string,
        shippingaddress:struct<
            address1:string,
            address2:string,
            city:string,
            state:string
        >,
        orders:array<
            struct<
                itemid:INT,
                orderdate:string
            >
        >
    >
)
ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'
LOCATION 's3://mybucket/myjsondata/';
```

Additional resources

For more information about working with JSON and nested JSON in Athena, see the following resources:

- Create tables in Amazon Athena from nested JSON and mappings using JSONSerDe (AWS Big Data Blog)
- I get errors when I try to read JSON data in Amazon Athena (AWS Knowledge Center article)
- hive-json-schema (GitHub) – Tool written in Java that generates CREATE TABLE statements from example JSON documents. The CREATE TABLE statements that are generated use the OpenX JSON Serde.
LazySimpleSerDe for CSV, TSV, and custom-delimited files

Specifying this SerDe is optional. This is the SerDe for data in CSV, TSV, and custom-delimited formats that Athena uses by default. This SerDe is used if you don’t specify any SerDe and only specify ROW FORMAT DELIMITED. Use this SerDe if your data does not have values enclosed in quotes.

For reference documentation about the LazySimpleSerDe, see the Hive SerDe section of the Apache Hive Developer Guide.

Library name

The Class library name for the LazySimpleSerDe is org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe. For information about the LazySimpleSerDe class, see LazySimpleSerDe.java on GitHub.com.

Ignoring headers

To ignore headers in your data when you define a table, you can use the skip.header.line.count table property, as in the following example.

TBLPROPERTIES ("skip.header.line.count"="1")

For examples, see the CREATE TABLE statements in Querying Amazon VPC flow logs (p. 474) and Querying Amazon CloudFront logs (p. 452).

Examples

The following examples show how to use the LazySimpleSerDe to create tables in Athena from CSV and TSV data. To deserialize custom-delimited files using this SerDe, follow the pattern in the examples but use the FIELDS TERMINATED BY clause to specify a different single-character delimiter. LazySimpleSerDe does not support multi-character delimiters.

- CSV Example (p. 324)
- TSV Example (p. 327)

Note

Replace myregion in s3://athena-examples-myregion/path/to/data/ with the region identifier where you run Athena, for example, s3://athena-examples-us-west-1/path/to/data/.

Note

The flight table data comes from Flights provided by US Department of Transportation, Bureau of Transportation Statistics. Desaturated from original.

CSV example

Use the CREATE TABLE statement to create an Athena table from the underlying data in CSV stored in Amazon S3.

```
CREATE EXTERNAL TABLE flight_delays_csv (yr INT,
```
quarter INT,
month INT,
dayofmonth INT,
dayofweek INT,
flightdate STRING,
uniquecarrier STRING,
airlineid INT,
carrier STRING,
tailnum STRING,
flightnum STRING,
originalAirportid INT,
originalPortseqid INT,
originCitymarketid INT,
origin STRING,
originCityname STRING,
originState STRING,
originStatefips STRING,
originWac INT,
destairportid INT,
destPortseqid INT,
destCitymarketid INT,
dest STRING,
destCityname STRING,
destState STRING,
destStatefips STRING,
destWac INT,
crsDepTime STRING,
depTime STRING,
depDelay INT,
depDelayMinutes INT,
depDelay15 INT,
depDelayGroups INT,
depTimeBlk STRING,
taxiOut INT,
wheelsoff STRING,
wheelson STRING,
taxiin INT,
crsArrTime INT,
arrTime STRING,
arrDelay INT,
arrDelayMinutes INT,
arrDelay15 INT,
arrDelayGroups INT,
arrTimeBlk STRING,
cancelled INT,
cancellationcode STRING,
diverted INT,
crs elapsedTime INT,
actualelapsedTime INT,
airTime INT,
flights INT,
distance INT,
distanceGroup INT,
carrierDelay INT,
weatherDelay INT,
nasDelay INT,
securityDelay INT,
lateAircraftDelay INT,
firstDepTime STRING,
totalAddgtime INT,
longestAddgtime INT,
divairportlandings INT,
divReacheddest INT,
divActualelapsedtime INT,
divarrdelay INT,
divdistance INT,
divlairport STRING,
divlairportid INT,
divlairportseqid INT,
divlwheelson STRING,
divltotalgtime INT,
divllongestgtime INT,
divlwheelsoff STRING,
divltailnum STRING,
div2airport STRING,
div2airportid INT,
div2airportseqid INT,
div2wheelson STRING,
div2totalgtime INT,
div2longestgtime INT,
div2wheelsoff STRING,
div2tailnum STRING,
div3airport STRING,
div3airportid INT,
div3airportseqid INT,
div3wheelson STRING,
div3totalgtime INT,
div3longestgtime INT,
div3wheelsoff STRING,
div3tailnum STRING,
div4airport STRING,
div4airportid INT,
div4airportseqid INT,
div4wheelson STRING,
div4totalgtime INT,
div4longestgtime INT,
div4wheelsoff STRING,
div4tailnum STRING,
div5airport STRING,
div5airportid INT,
div5airportseqid INT,
div5wheelson STRING,
div5totalgtime INT,
div5longestgtime INT,
div5wheelsoff STRING,
div5tailnum STRING
)
PARTITIONED BY (year STRING)
ROW FORMAT DELIMITED
   FIELDS TERMINATED BY ','
   ESCAPED BY '\'
   LINES TERMINATED BY '\n'
LOCATION 's3://athena-examples-myregion/flight/csv/';

Run MSCK REPAIR TABLE to refresh partition metadata each time a new partition is added to this table:

MSCK REPAIR TABLE flight_delays_csv;

Query the top 10 routes delayed by more than 1 hour:

SELECT origin, dest, count(*) as delays
FROM flight_delays_csv
WHERE depdelayminutes > 60
GROUP BY origin, dest
ORDER BY 3 DESC
LIMIT 10;
TSV example

Use a CREATE TABLE statement to create an Athena table from the TSV data stored in Amazon S3. Use ROW FORMAT DELIMITED and specify the tab field delimiter, line separator, and escape character as follows:

```
... 
ROW FORMAT DELIMITED 
FIELDS TERMINATED BY '\t'
ESCAPED BY '\\\'
LINES TERMINATED BY '\n'
... 
```

An example CREATE TABLE statement follows.

```
CREATE EXTERNAL TABLE flight_delays_tsv (
  yr INT,
  quarter INT,
  month INT,
  dayofmonth INT,
  dayofweek INT,
  flightdate STRING,
  uniquecarrier STRING,
  airlineid INT,
  carrier STRING,
  tailnum STRING,
  flightnum STRING,
  originairportid INT,
  originairportseqid INT,
  origincitymarketid INT,
  origin STRING,
  origincityname STRING,
  originstate STRING,
  originstatefips STRING,
  originwac INT,
  destairportid INT,
  destairportseqid INT,
  destcitymarketid INT,
  dest STRING,
  destcityname STRING,
  deststate STRING,
  deststatefips STRING,
  destwac INT,
  crsdeptime STRING,
  depdelay STRING,
  depdelayminutes INT,
  depdelay15 INT,
  departrdelaygroups INT,
  deptimeblk STRING,
  taxilout INT,
  wheelsoff STRING,
  wheelson STRING,
  taxii INT,
  crsarrtime INT,
  arftime STRING,
  arrdelay INT,
  arrdelayminutes INT,
  arrdel15 INT,
  arrivaldelaygroups INT,
  arrtimeblk STRING,
```
cancelled INT,
cancellationcode STRING,
diverted INT,
crseelapsedtime INT,
actualelapsedtime INT,
airtime INT,
flights INT,
distance INT,
distancegroup INT,
carrierdelay INT,
weatherdelay INT,
nasdelay INT,
securitydelay INT,
lateaircraftdelay INT,
firstdeptime STRING,
totaladdgtime INT,
longestaddgtime INT,
divairportlandings INT,
diveacheddest INT,
divactualelapsedtime INT,
divrdelay INT,
divdistance INT,
div1airport STRING,
div1airportid INT,
div1airportseqid INT,
div1wheelson STRING,
div1totalgtime INT,
div1longestgtime INT,
div1wheelsoff STRING,
div1tailnum STRING,
div2airport STRING,
div2airportid INT,
div2airportseqid INT,
div2wheelson STRING,
div2totalgtime INT,
div2longestgtime INT,
div2wheelsoff STRING,
div2tailnum STRING,
div3airport STRING,
div3airportid INT,
div3airportseqid INT,
div3wheelson STRING,
div3totalgtime INT,
div3longestgtime INT,
div3wheelsoff STRING,
div3tailnum STRING,
div4airport STRING,
div4airportid INT,
div4airportseqid INT,
div4wheelson STRING,
div4totalgtime INT,
div4longestgtime INT,
div4wheelsoff STRING,
div4tailnum STRING,
div5airport STRING,
div5airportid INT,
div5airportseqid INT,
div5wheelson STRING,
div5totalgtime INT,
div5longestgtime INT,
div5wheelsoff STRING,
div5tailnum STRING
)
PARTITIONED BY (year STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY 't'
ESCAPED BY '\\'
LINES TERMINATED BY '\n'
LOCATION 's3://athena-examples-myregion/flight/tsv/';

Run MSCK REPAIR TABLE to refresh partition metadata each time a new partition is added to this table:

```
MSCK REPAIR TABLE flight_delays_tsv;
```

Query the top 10 routes delayed by more than 1 hour:

```
SELECT origin, dest, count(*) as delays
FROM flight_delays_tsv
WHERE depdelayminutes > 60
GROUP BY origin, dest
ORDER BY 3 DESC
LIMIT 10;
```

**Note**
The flight table data comes from Flights provided by US Department of Transportation, Bureau of Transportation Statistics. Desaturated from original.

#### OpenCSVSerDe for processing CSV

When you create an Athena table for CSV data, determine the SerDe to use based on the types of values your data contains:

- If your data contains values enclosed in double quotes ("), you can use the OpenCSV SerDe to deserialize the values in Athena. If your data does not contain values enclosed in double quotes ("), you can omit specifying any SerDe. In this case, Athena uses the default LazySimpleSerDe. For information, see LazySimpleSerDe for CSV, TSV, and custom-delimited files (p. 324).
- If your data has UNIX numeric TIMESTAMP values (for example, 1579059880000), use the OpenCSVSerDe. If your data uses the java.sql.Timestamp format, use the LazySimpleSerDe.

#### CSV SerDe (OpenCSVSerDe)

The OpenCSV SerDe has the following characteristics for string data:

- Uses double quotes (") as the default quote character, and allows you to specify separator, quote, and escape characters, such as:

  ```
  WITH SERDEPROPERTIES ("separatorChar" = ",", "quoteChar" = "\", "escapeChar" = "\\")
  ```

- Cannot escape \t or \n directly. To escape them, use "escapeChar" = "\\". See the example in this topic.
- Does not support embedded line breaks in CSV files.

For data types other than STRING, the OpenCSVSerDe behaves as follows:

- Recognizes BOOLEAN, BIGINT, INT, and DOUBLE data types.
- Does not recognize empty or null values in columns defined as a numeric data type, leaving them as string. One workaround is to create the column with the null values as string and then use CAST to convert the field in a query to a numeric data type, supplying a default value of 0 for nulls. For more
information, see When I query CSV data in Athena, I get the error HIVE_BAD_DATA: Error parsing field value in the AWS Knowledge Center.

- For columns specified with the timestamp data type in your CREATE TABLE statement, recognizes TIMESTAMP data if it is specified in the UNIX numeric format in milliseconds, such as 1579059880000.
  - The OpenCSVSerDe does not support TIMESTAMP in the JDBC-compliant java.sql.Timestamp format, such as "YYYY-MM-DD HH:MM:SS.SSSSSSSSSS" (9 decimal place precision).
- For columns specified with the DATE data type in your CREATE TABLE statement, recognizes values as dates if the values represent the number of days that elapsed since January 1, 1970. For example, the value 18276 in a column with the date data type renders as 2020-01-15 when queried. In this UNIX format, each day is considered to have 86,400 seconds.
  - The OpenCSVSerDe does not support DATE in any other format directly. To process timestamp data in other formats, you can define the column as string and then use time conversion functions to return the desired results in your SELECT query. For more information, see the article When I query a table in Amazon Athena, the TIMESTAMP result is empty in the AWS knowledge center.
- To further convert columns to the desired type in a table, you can create a view over the table and use CAST to convert to the desired type.

**Example: Using the TIMESTAMP type and DATE type specified in the UNIX numeric format.**

Consider the following three columns of comma-separated data. The values in each column are enclosed in double quotes.

```
"unixvalue creationdate 18276 creationdatetime 1579059880000","18276","1579059880000"
```

The following statement creates a table in Athena from the specified Amazon S3 bucket location.

```
CREATE EXTERNAL TABLE IF NOT EXISTS testtimestamp1(`profile_id` string, `creationdate` date, `creationdatetime` timestamp)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
LOCATION 's3://DOC-EXAMPLE-BUCKET'
```

Next, run the following query:

```
SELECT * FROM testtimestamp1
```

The query returns the following result, showing the date and time data:

```
profile_id creationdate creationdatetime
unixvalue creationdate 18276 creationdatetime 1579146280000 2020-01-15 03:44:40.000
```

**Example: Escaping \t or \n**

Consider the following test data:

```
" \t\t\n 123 \t\t\n ",abc
" 456 ",xyz
```

The following statement creates a table in Athena, specifying that "escapeChar" = "\".

```
" \t\t\n 123 \t\t\n ",abc
" 456 ",xyz
```

330
CREATE EXTERNAL TABLE test1 (  
f1 string,  
s2 string)  
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'  
WITH SERDEPROPERTIES ("separatorChar" = ",", "quoteChar" = "\"", "escapeChar" = "\\")  
LOCATION 's3://DOC-EXAMPLE-BUCKET/dataset/test1/'

Next, run the following query:

SELECT * FROM test1;

It returns this result, correctly escaping \t or \n:

<table>
<thead>
<tr>
<th>f1</th>
<th>s2</th>
</tr>
</thead>
<tbody>
<tr>
<td>123 \t</td>
<td>\t abc</td>
</tr>
<tr>
<td>456 \t</td>
<td>\t xyz</td>
</tr>
</tbody>
</table>

SerDe name

CSV SerDe

Library name

To use this SerDe, specify its fully qualified class name after ROW FORMAT SERDE. Also specify the delimiters inside SERDEPROPERTIES, as follows:

```
...  
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'  
WITH SERDEPROPERTIES (  
  "separatorChar" = ",",  
  "quoteChar" = "\",  
  "escapeChar" = "\\"  
)
```

Ignoring headers

To ignore headers in your data when you define a table, you can use the skip.header.line.count table property, as in the following example.

```
TBLPROPERTIES ("skip.header.line.count"="1")
```

For examples, see the CREATE TABLE statements in Querying Amazon VPC flow logs (p. 474) and Querying Amazon CloudFront logs (p. 452).

Example

This example presumes data in CSV saved in s3://DOC-EXAMPLE-BUCKET/mycsv/ with the following contents:

```
"a1", "a2", "a3", "a4"  
"1", "2", "abc", "def"  
"a1", "a1", "abc3", "ab4"
```

Use a CREATE TABLE statement to create an Athena table based on the data. Reference the OpenCSVSerDe class after ROW FORMAT SERDE and specify the character separator, quote character, and escape character in WITH SERDEPROPERTIES, as in the following example.
CREATE EXTERNAL TABLE myopencsvtable (  
col1 string,  
col2 string,  
col3 string,  
col4 string  
)  
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'  
WITH SERDEPROPERTIES (  
'separatorChar' = ',',  
'quoteChar' = '"',  
'escapeChar' = '\'  
)  
STORED AS TEXTFILE  
LOCATION 's3://DOC-EXAMPLE-BUCKET/mycsv/';

Query all values in the table:

SELECT * FROM myopencsvtable;

The query returns the following values:

<table>
<thead>
<tr>
<th>col1</th>
<th>col2</th>
<th>col3</th>
<th>col4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>a2</td>
<td>a3</td>
<td>a4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>abc</td>
<td>def</td>
</tr>
<tr>
<td>a</td>
<td>a1</td>
<td>abc3</td>
<td>ab4</td>
</tr>
</tbody>
</table>

**ORC SerDe**

**SerDe name**

OrcSerDe

**Library name**

This library uses the Apache Hive OrcSerde.java class for data in the ORC format. It passes the object from ORC to the reader and from ORC to the writer.

**Examples**

**Note**

Replace myregion in s3://athena-examples-myregion/path/to/data/ with the region identifier where you run Athena, for example, s3://athena-examples-us-west-1/path/to/data/.

The following example creates a table for the flight delays data in ORC. The table includes partitions:

DROP TABLE flight_delays_orc;
CREATE EXTERNAL TABLE flight_delays_orc (  
yr INT,  
quarter INT,  
month INT,  
dayofmonth INT,  
dayofweek INT,  
flightdate STRING,  
uniquecarrier STRING,  
 flights in myregion with partitioning.
airlineid INT,
carrier STRING,
tailnum STRING,
flightnum STRING,
originalairportid INT,
originalairportseqid INT,
originairportmarketid INT,
origin STRING,
originatymarktnam STRING,
originstate STRING,
originstatefips STRING,
originstatetmename STRING,
originwac INT,
destinationairportid INT,
destinationairportseqid INT,
destcitymarketid INT,
dest STRING,
destcitymarktnam STRING,
deststate STRING,
deststatefips STRING,
deststatetmename STRING,
destwac INT,
crsdeptime STRING,
deptime STRING,
dep delay INT,
dep delay minutes INT,
dep delay IS INT,
departuredelaygroups INT,
depdelayblk STRING,
taxiout INT,
wheels off STRING,
wheels on STRING,
taxiin INT,
crsarrtime INT,
arrtime STRING,
arr delay INT,
arr delay minutes INT,
arr delay IS INT,
arrivaldelaygroups INT,
arrdelayblk STRING,
cancelled INT,
cancellationcode STRING,
diverted INT,
crs elapsedtime INT,
actual elapsedtime INT,
airtime INT,
flights INT,
distance INT,
distance group INT,
carrer delay INT,
weather delay INT,
nas delay INT,
security delay INT,
late aircraft delay INT,
first dep time STRING,
total add time INT,
longest add time INT,
div airport landings INT,
div reached dest INT,
div actual elapsedtime INT,
div arr delay INT,
div distance INT,
div 1 airport STRING,
div 1 airport id INT,
div 1 wheelson STRING,
div1totalgtime INT,
div1longestgtime INT,
div1wheelsoff STRING,
div1tailnum STRING,
div2airport STRING,
div2airportid INT,
div2airportseqid INT,
div2wheelson STRING,
div2totalgtime INT,
div2longestgtime INT,
div2wheelsoff STRING,
div2tailnum STRING,
div3airport STRING,
div3airportid INT,
div3airportseqid INT,
div3wheelson STRING,
div3totalgtime INT,
div3longestgtime INT,
div3wheelsoff STRING,
div3tailnum STRING,
div4airport STRING,
div4airportid INT,
div4airportseqid INT,
div4wheelson STRING,
div4totalgtime INT,
div4longestgtime INT,
div4wheelsoff STRING,
div4tailnum STRING,
div5airport STRING,
div5airportid INT,
div5airportseqid INT,
div5wheelson STRING,
div5totalgtime INT,
div5longestgtime INT,
div5tailnum STRING
)
PARTITIONED BY (year String)
STORED AS ORC
LOCATION 's3://athena-examples-myregion/flight/orc/'
tblproperties ("orc.compress"="ZLIB");

Run the `MSCK REPAIR TABLE` statement on the table to refresh partition metadata:

```
MSCK REPAIR TABLE flight_delays_orc;
```

Use this query to obtain the top 10 routes delayed by more than 1 hour:

```
SELECT origin, dest, count(*) as delays
FROM flight_delays_orc
WHERE depdelayminutes > 60
GROUP BY origin, dest
ORDER BY 3 DESC
LIMIT 10;
```

---

**Parquet SerDe**

**SerDe name**

ParquetHiveSerDe is used for data stored in Parquet format.
Note
To convert data into Parquet format, you can use CREATE TABLE AS SELECT (CTAS) (p. 744) queries. For more information, see Creating a table from query results (CTAS) (p. 281), Examples of CTAS queries (p. 286) and Using CTAS and INSERT INTO for ETL and data analysis (p. 290).

Library name
Athena uses the following class when it needs to deserialize data stored in Parquet:
org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe

Example: Querying a file stored in parquet

Note
Replace myregion in s3://athena-examples-myregion/path/to/data/ with the region identifier where you run Athena, for example, s3://athena-examples-us-west-1/path/to/data/.

Use the following CREATE TABLE statement to create an Athena table from the underlying data in CSV stored in Amazon S3 in Parquet:

```
CREATE EXTERNAL TABLE flight_delays_pq (
   yr INT,
   quarter INT,
   month INT,
   dayofmonth INT,
   dayofweek INT,
   flightdate STRING,
   uniquecarrier STRING,
   airlineid INT,
   carrier STRING,
   tailnum STRING,
   flightnum STRING,
   originairportid INT,
   originairportseqid INT,
   origincitymarketid INT,
   origin STRING,
   origincityname STRING,
   originstate STRING,
   originstatefips STRING,
   originstatename STRING,
   originwac INT,
   destairportid INT,
   destairportseqid INT,
   destcitymarketid INT,
   dest STRING,
   destcityname STRING,
   deststate STRING,
   deststatefips STRING,
   deststatename STRING,
   destwac INT,
   crsdeptime STRING,
   deptime STRING,
   depdelay INT,
   depdelayminutes INT,
   depdell5 INT,
   departuredelaygroups INT,
   depdelayblks INT,
   taxiout INT,
   wheelsoff STRING,
   wheelson STRING,
   taxiln INT,
   crsarrivaltime INT,
```
arrrime STRING,
arrrdelay INT,
arrrdelayminutes INT,
arrrdelay15 INT,
arrrivaldelaygroups INT,
arrrtimeblk STRING,
cancelled INT,
cancellationcode STRING,
diverted INT,
crselapsedtime INT,
actualelapsedtime INT,
airtime INT,
flights INT,
distance INT,
distancegroup INT,
carrierged INT,
weatherdelay INT,
nasdelay INT,
securitydelay INT,
lateaircraftdelay INT,
firstdeptime STRING,
totaladdgtime INT,
longestaddgtime INT,
divairportlandings INT,
divreacheddest INT,
divactualelapsedtime INT,
divarrdelay INT,
divdistance INT,
divairport STRING,
divairportid INT,
divairportseqid INT,
div1wheelson STRING,
div1totalgtime INT,
div1longestgtime INT,
div1wheelsoff STRING,
div1tailnum STRING,
div2airport STRING,
div2airportid INT,
div2airportseqid INT,
div2wheelson STRING,
div2totalgtime INT,
div2longestgtime INT,
div2wheelsoff STRING,
div2tailnum STRING,
div3airport STRING,
div3airportid INT,
div3airportseqid INT,
div3wheelson STRING,
div3totalgtime INT,
div3longestgtime INT,
div3wheelsoff STRING,
div3tailnum STRING,
div4airport STRING,
div4airportid INT,
div4airportseqid INT,
div4wheelson STRING,
div4totalgtime INT,
div4longestgtime INT,
div4wheelsoff STRING,
div4tailnum STRING,
div5airport STRING,
div5airportid INT,
div5airportseqid INT,
div5wheelson STRING,
div5totalgtime INT,
div5longestgtime INT,
Run the `MSCK REPAIR TABLE` statement on the table to refresh partition metadata:

```sql
MSCK REPAIR TABLE flight_delays_pq;
```

Query the top 10 routes delayed by more than 1 hour:

```sql
SELECT origin, dest, count(*) as delays
FROM flight_delays_pq
WHERE depdelayminutes > 60
GROUP BY origin, dest
ORDER BY 3 DESC
LIMIT 10;
```

**Note**
The flight table data comes from Flights provided by US Department of Transportation, Bureau of Transportation Statistics. Desaturated from original.

### Regex SerDe

The Regex SerDe uses a regular expression (regex) to deserialize data by extracting regex groups into table columns.

If a row in the data does not match the regex, then all columns in the row are returned as `NULL`. If a row matches the regex but has fewer groups than expected, the missing groups are `NULL`. If a row in the data matches the regex but has more columns than groups in the regex, the additional columns are ignored.

For more information, see the `Class RegexSerDe` in the Apache Hive documentation.

### SerDe name

RegexSerDe

### Library name

RegexSerDe

### Examples

The following example creates a table from CloudFront logs using the RegExSerDe. Replace `myregion` in `s3://athena-examples-myregion/cloudfront/plaintext/` with the region identifier where you run Athena (for example, `s3://athena-examples-us-west-1/cloudfront/plaintext/`).

```sql
CREATE EXTERNAL TABLE IF NOT EXISTS cloudfront_logs (  
  Date DATE,
  Time STRING,
  Location STRING,
  Bytes INT,
  RequestIP STRING,
  Method STRING,
) PARTITIONED BY (year STRING)
STORED AS PARQUET
LOCATION 's3://athena-examples-myregion/cloudfront/plaintext/'
tblproperties ("parquet.compression"="SNAPPY");
```
Host STRING,
Uri STRING,
Status INT,
Referrer STRING,
oS STRING,
Browser STRING,
BrowserVersion STRING
) ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES ("input.regex" = "^(?!#)(([^ \ ]+)\s+([^ \ ]+)\s+([^ \ ]+)\s+([^ \ ]+)\s+([^ \ ]+)\s+([^ \ ]+)\s+([^ \ ]+)\s+([^ \ ]+)\s+([^ \ ]+)\s+([^ \ ]+)\s+([^ \ ]+)\s+(\[\^\]%)\s+(.*)$"
) LOCATION 's3://athena-examples-myregion/cloudfront/plaintext/';
Running SQL queries using Amazon Athena

You can run SQL queries using Amazon Athena on data sources that are registered with the AWS Glue Data Catalog and data sources such as Hive metastores and Amazon DocumentDB instances that you connect to using the Athena Federated Query feature. For more information about working with data sources, see Connecting to data sources (p. 21). When you run a Data Definition Language (DDL) query that modifies schema, Athena writes the metadata to the metastore associated with the data source. In addition, some queries, such as CREATE TABLE AS and INSERT INTO can write records to the dataset—for example, adding a CSV record to an Amazon S3 location. When you run a query, Athena saves the results of a query in a query result location that you specify. This allows you to view query history and to download and view query results sets.

This section provides guidance for running Athena queries on common data sources and data types using a variety of SQL statements. General guidance is provided for working with common structures and operators—for example, working with arrays, concatenating, filtering, flattening, and sorting. Other examples include queries for data in tables with nested structures and maps, tables based on JSON-encoded datasets, and datasets associated with AWS services such as AWS CloudTrail logs and Amazon EMR logs. Comprehensive coverage of standard SQL usage is beyond the scope of this documentation. For more information about SQL, refer to the Trino and Presto language references.

Topics
- Viewing execution plans for SQL queries (p. 340)
- Working with query results, recent queries, and output files (p. 344)
- Reusing query results (p. 354)
- Viewing statistics and execution details for completed queries (p. 357)
- Working with views (p. 361)
- Using saved queries (p. 368)
- Using parameterized queries (p. 370)
- Handling schema updates (p. 376)
- Querying arrays (p. 385)
- Querying geospatial data (p. 402)
- Using Athena to query Apache Hudi datasets (p. 420)
- Querying JSON (p. 425)
- Using Machine Learning (ML) with Amazon Athena (p. 432)
- Querying with user defined functions (p. 434)
- Querying across regions (p. 442)
- Querying AWS Glue Data Catalog (p. 442)
- Querying AWS service logs (p. 447)
- Querying web server logs stored in Amazon S3 (p. 489)

For considerations and limitations, see Considerations and limitations for SQL queries in Amazon Athena (p. 764).
Viewing execution plans for SQL queries

You can use the Athena query editor to see graphical representations of how your query will be run. When you enter a query in the editor and choose the **Explain** option, Athena uses an **EXPLAIN** (p. 703) SQL statement on your query to create two corresponding graphs: a distributed execution plan and a logical execution plan. You can use these graphs to analyze, troubleshoot, and improve the efficiency of your queries.

**To view execution plans for a query**

1. Enter your query in the Athena query editor, and then choose **Explain**.

![Example SQL query]

The **Distributed plan** tab shows you the execution plan for your query in a distributed environment. A distributed plan has processing fragments or **stages**. Each stage has a zero-based index number and is processed by one or more nodes. Data can be exchanged between nodes.
2. To navigate the graph, use the following options:
   - To zoom in or out, scroll the mouse, or use the magnifying icons.
   - To adjust the graph to fit the screen, choose the **Zoom to fit** icon.
   - To move the graph around, drag the mouse pointer.

3. To see details for a stage, choose the stage.
4. To see the stage details full width, choose the expand icon at the top right of the details pane.
5. To see more detail, expand one or more items in the operator tree. For information about distributed plan fragments, see EXPLAIN statement output types (p. 712).
Important
Currently, some partition filters may not be visible in the nested operator tree graph even though Athena does apply them to your query. To verify the effect of such filters, run `EXPLAIN` (p. 703) or `EXPLAIN ANALYZE` (p. 703) on your query and view the results.

6. Choose the **Logical plan** tab. The graph shows the logical plan for running your query. For information about operational terms, see *Understanding Athena EXPLAIN statement results* (p. 712).

7. To export a plan as an SVG or PNG image, or as JSON text, choose **Export**.
Working with query results, recent queries, and output files

Amazon Athena automatically stores query results and metadata information for each query that runs in a query result location that you can specify in Amazon S3. If necessary, you can access the files in this location to work with them. You can also download query result files directly from the Athena console.

To set up an Amazon S3 query result location for the first time, see Specifying a query result location using the Athena console (p. 344).

Output files are saved automatically for every query that runs. To access and view query output files, IAM principals (users and roles) need permission to the Amazon S3 GetObject action for the query result location, as well as permission for the Athena GetQueryResults action. The query result location can be encrypted. If the location is encrypted, users must have the appropriate key permissions to encrypt and decrypt the query result location.

Important
IAM principals with permission to the Amazon S3 GetObject action for the query result location are able to retrieve query results from Amazon S3 even if permission to the Athena GetQueryResults action is denied.

Specifying a query result location

The query result location that Athena uses is determined by a combination of workgroup settings and client-side settings. Client-side settings are based on how you run the query.

- If you run the query using the Athena console, the Query result location entered under Settings in the navigation bar determines the client-side setting.
- If you run the query using the Athena API, the OutputLocation parameter of the StartQueryExecution action determines the client-side setting.
- If you use the ODBC or JDBC drivers to run queries, the S3OutputLocation property specified in the connection URL determines the client-side setting.

Important
When you run a query using the API or using the ODBC or JDBC driver, the console setting does not apply.

Each workgroup configuration has an Override client-side settings option that can be enabled. When this option is enabled, the workgroup settings take precedence over the applicable client-side settings when an IAM principal associated with that workgroup runs the query.

Specifying a query result location using the Athena console

Before you can run a query, a query result bucket location in Amazon S3 must be specified, or you must use a workgroup that has specified a bucket and whose configuration overrides client settings.
To specify a client-side setting query result location using the Athena console

1. Switch (p. 643) to the workgroup for which you want to specify a query results location. The name of the default workgroup is primary.
2. From the navigation bar, choose Settings.
3. From the navigation bar, choose Manage.
4. For Manage settings, do one of the following:
   - In the Location of query result box, enter the path to the bucket that you created in Amazon S3 for your query results. Prefix the path with s3://.
   - Choose Browse S3, choose the Amazon S3 bucket that you created for your current Region, and then choose Choose.

   **Note**
   If you are using a workgroup that specifies a query result location for all users of the workgroup, the option to change the query result location is unavailable.

5. (Optional) For Expected bucket owner, enter the ID of the AWS account that you expect to be the owner of the output location bucket. This is an added security measure. If the account ID of the bucket owner does not match the ID that you specify here, attempts to output to the bucket will fail. For in-depth information, see Verifying bucket ownership with bucket owner condition in the Amazon S3 User Guide.

   **Note**
   The expected bucket owner setting applies only to the Amazon S3 output location that you specify for Athena query results. It does not apply to other Amazon S3 locations like data source locations in external Amazon S3 buckets, CTAS and INSERT INTO destination table locations, UNLOAD statement output locations, operations to spill buckets for federated queries, or SELECT queries run against a table in another account.

6. (Optional) Choose Encrypt query results if you want to encrypt the query results stored in Amazon S3. For more information about encryption in Athena, see Encryption at rest (p. 521).

7. (Optional) Choose Assign bucket owner full control over query results to grant full control access over query results to the bucket owner when ACLs are enabled for the query result bucket. For example, if your query result location is owned by another account, you can grant ownership and full control over your query results to the other account. For more information, see Controlling ownership of objects and disabling ACLs for your bucket in the Amazon S3 User Guide.

8. Choose Save.

Previously created default locations

Previously in Athena, if you ran a query without specifying a value for Query result location, and the query result location setting was not overridden by a workgroup, Athena created a default location for you. The default location was aws-athena-query-results-MyAcctID-MyRegion, where MyAcctID was the Amazon Web Services account ID of the IAM principal that ran the query, and MyRegion was the region where the query ran (for example, us-west-1.)

Now, before you can run an Athena query in a region in which your account hasn't used Athena previously, you must specify a query result location, or use a workgroup that overrides the query result location setting. While Athena no longer creates a default query results location for you, previously created default aws-athena-query-results-MyAcctID-MyRegion locations remain valid and you can continue to use them.
Specifying a query result location using a workgroup

You specify the query result location in a workgroup configuration using the AWS Management Console, the AWS CLI, or the Athena API.

When using the AWS CLI, specify the query result location using the `OutputLocation` parameter of the `--configuration` option when you run the `aws athena create-work-group` or `aws athena update-work-group` command.

To specify the query result location for a workgroup using the Athena console

1. If the console navigation pane is not visible, choose the expansion menu on the left.
2. In the navigation pane, choose **Workgroups**.
3. In the list of workgroups, choose the link of the workgroup that you want to edit.
4. Choose **Edit**.
5. For **Query result location and encryption**, do one of the following:
   - In the **Location of query result** box, enter the path to a bucket in Amazon S3 for your query results. Prefix the path with `s3://`.
   - Choose **Browse S3**, choose the Amazon S3 bucket for your current Region that you want to use, and then choose **Choose**.
6. (Optional) For **Expected bucket owner**, enter the ID of the AWS account that you expect to be the owner of the output location bucket. This is an added security measure. If the account ID of the bucket owner does not match the ID that you specify here, attempts to output to the bucket will fail. For in-depth information, see [Verifying bucket ownership with bucket owner condition](https://docs.aws.amazon.com/AmazonS3/latest/userguide/bucket-owner-encryption.html) in the *Amazon S3 User Guide*.

   **Note**
   
   The expected bucket owner setting applies only to the Amazon S3 output location that you specify for Athena query results. It does not apply to other Amazon S3 locations like data source locations in external Amazon S3 buckets, CTAS and INSERT INTO destination table locations, UNLOAD statement output locations, operations to spill buckets for federated queries, or SELECT queries run against a table in another account.

7. (Optional) Choose **Encrypt query results** if you want to encrypt the query results stored in Amazon S3. For more information about encryption in Athena, see [Encryption at rest](https://docs.aws.amazon.com/AmazonAthena/latest/ug/encryption-at-rest.html) (p. 521).
8. (Optional) Choose **Assign bucket owner full control over query results** to grant full control access over query results to the bucket owner when ACLs are enabled for the query result bucket. For example, if your query result location is owned by another account, you can grant ownership and full control over your query results to the other account.

   If the bucket's S3 Object Ownership setting is **Bucket owner preferred**, the bucket owner also owns all query result objects written from this workgroup. For example, if an external account's workgroup enables this option and sets its query result location to your account's Amazon S3 bucket which has
an S3 Object Ownership setting of **Bucket owner preferred**, you own and have full control access over the external workgroup's query results.

Selecting this option when the query result bucket's S3 Object Ownership setting is **Bucket owner enforced** has no effect. For more information, see "Controlling ownership of objects and disabling ACLs for your bucket" in the *Amazon S3 User Guide*.

9. If you want to require all users of the workgroup to use the query results location that you specified, scroll down to the **Settings** section and select **Override client-side settings**.

10. Choose **Save changes**.

---

**Downloading query results files using the Athena console**

You can download the query results CSV file from the query pane immediately after you run a query. You can also download query results from recent queries from the **Recent queries** tab.

**Note**

Athena query result files are data files that contain information that can be configured by individual users. Some programs that read and analyze this data can potentially interpret some of the data as commands (CSV injection). For this reason, when you import query results CSV data to a spreadsheet program, that program might warn you about security concerns. To keep your system secure, you should always choose to disable links or macros from downloaded query results.

**To run a query and download the query results**

1. Enter your query in the query editor and then choose **Run**.

   When the query finishes running, the **Results** pane shows the query results.

2. To download a CSV file of the query results, choose **Download results** above the query results pane. Depending on your browser and browser configuration, you may need to confirm the download.

![](image)

**To download a query results file for an earlier query**

1. Choose **Recent queries**.
2. Use the search box to find the query, select the query, and then choose **Download results**. Queries are retained for 45 days.
**Viewing recent queries**

You can use the Athena console to see which queries succeeded or failed, and view error details for the queries that failed. Athena keeps a query history for 45 days.

**To view recent queries in the Athena console**

2. Choose **Recent queries**. The **Recent queries** tab shows information about each query that ran.
3. To open a query statement in the query editor, choose the query's execution ID.
4. To see the details for a query that failed, choose the **Failed** link for the query.
5. To configure options for the Recent queries tab like columns to display and text wrapping, choose the options button (gear icon).

6. In the Preferences dialog box, choose the number of rows per page, line wrapping behavior, and columns to display.
Preferences

Select rows per page
- 10 queries
- 20 queries

□ Wrap lines
Wraps long lines to show all the text

Select visible content

Properties

- Execution ID
- Query
- Start time
- Run time
- Status
- Data scanned
- Query engine version used
- Encryption

[ Confirm ] [ Cancel ]
7. Choose Confirm.

**Keeping query history longer than 45 days**

If you want to keep the query history longer than 45 days, you can retrieve the query history and save it to a data store such as Amazon S3. To automate this process, you can use Athena and Amazon S3 API actions and CLI commands. The following procedure summarizes these steps.

**To retrieve and save query history programmatically**

1. Use Athena `ListQueryExecutions` API action or the `list-query-executions` CLI command to retrieve the query IDs.
2. Use the Athena `GetQueryExecution` API action or the `get-query-execution` CLI command to retrieve information about each query based on its ID.
3. Use the Amazon S3 `PutObject` API action or the `put-object` CLI command to save the information in Amazon S3.

**Finding query output files in Amazon S3**

Query output files are stored in sub-folders on Amazon S3 in the following path pattern unless the query occurs in a workgroup whose configuration overrides client-side settings. When workgroup configuration overrides client-side settings, the query uses the results path specified by the workgroup.

```
QueryResultsLocationInS3/[/QueryName|Unsaved]/yyyy/mm/dd
```

- **QueryResultsLocationInS3** is the query result location specified either by workgroup settings or client-side settings. For more information, see the section called “Specifying a query result location” (p. 344) later in this document.
- The following sub-folders are created only for queries run from the console whose results path has not been overridden by workgroup configuration. Queries that run from the AWS CLI or using the Athena API are saved directly to the `QueryResultsLocationInS3`.
  - **QueryName** is the name of the query for which the results are saved. If the query ran but wasn't saved, Unsaved is used.
  - **yyyy/mm/dd** is the date that the query ran.

Files associated with a `CREATE TABLE AS SELECT` query are stored in a `tables` sub-folder of the above pattern.

**Identifying query output files**

Files are saved to the query result location in Amazon S3 based on the name of the query, the query ID, and the date that the query ran. Files for each query are named using the `QueryID`, which is a unique identifier that Athena assigns to each query when it runs.

The following file types are saved:

<table>
<thead>
<tr>
<th>File type</th>
<th>File naming patterns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query results files</td>
<td><code>QueryID.csv</code></td>
<td>DML query results files are saved in comma-separated values (CSV) format.</td>
</tr>
<tr>
<td></td>
<td><code>QueryID.txt</code></td>
<td></td>
</tr>
</tbody>
</table>

352
### Finding query output files in Amazon S3

<table>
<thead>
<tr>
<th>File type</th>
<th>File naming patterns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DDL query results</strong></td>
<td></td>
<td>DDL query results are saved as plain text files. You can download results files from the console from the Results pane when using the console or from the query History. For more information, see Downloading query results files using the Athena console (p. 347).</td>
</tr>
<tr>
<td><strong>Query metadata files</strong></td>
<td><code>QueryID.csv.metadata</code></td>
<td>DML and DDL query metadata files are saved in binary format and are not human readable. The file extension corresponds to the related query results file. Athena uses the metadata when reading query results using the GetQueryResults action. Although these files can be deleted, we do not recommend it because important information about the query is lost.</td>
</tr>
<tr>
<td><strong>Data manifest files</strong></td>
<td><code>QueryID-manifest.csv</code></td>
<td>Data manifest files are generated to track files that Athena creates in Amazon S3 data source locations when an INSERT INTO (p. 697) query runs. If a query fails, the manifest also tracks files that the query intended to write. The manifest is useful for identifying orphaned files resulting from a failed query.</td>
</tr>
</tbody>
</table>

### Using the AWS CLI to identify query output location and files

To use the AWS CLI to identify the query output location and result files, run the `aws athena get-query-execution` command, as in the following example. Replace `abc1234d-5efg-67hi-jklm-89n0op12qr34` with the query ID.

```bash
aws athena get-query-execution --query-execution-id abc1234d-5efg-67hi-jklm-89n0op12qr34
```

The command returns output similar to the following. For descriptions of each output parameter, see get-query-execution in the AWS CLI Command Reference.

```json
{
  "QueryExecution": {
    "Status": {
      "SubmissionDateTime": 1565649050.175,
      "State": "SUCCEEDED",
      "CompletionDateTime": 1565649056.6229999
    }
  }
}
```
Reusing query results

When you re-run a query in Athena, you can optionally choose to reuse the last stored query result. This option can increase performance and reduce costs in terms of the number of bytes scanned. Reusing query results is useful if, for example, you know that the results will not change within a given time frame.

You can specify a maximum age for reusing query results. Athena uses the stored result as long as it is not older than the age that you specify.

Key features

- Reusing query results is a per-query, opt-in feature. You can enable query result reuse on a per query basis.
- The maximum age for reusing query results can be specified in minutes, hours, or days. The maximum age specifiable is the equivalent of 7 days regardless of the time unit used. The default is 60 minutes.
- When you enable result reuse for a query, Athena looks for a previous execution of the query within the same workgroup. If Athena finds corresponding stored query results, it does not rerun the query, but points to the previous result location or fetches data from it.
- For any query that enables the results reuse option, Athena reuses the last query result saved to the workgroup folder only when all the following conditions are true:
  - The query string is an exact match.
  - The database and the catalog name match.
  - The previous result is not older than the maximum age specified, or not older than 60 minutes if a maximum age has not been specified.
  - Athena only reuses an execution that has the exact same result configuration as the current execution.
  - You have access to all the tables referenced in the query.
  - You have access to the S3 file location where the previous result is stored.

If any of these conditions are not met, Athena runs the query without using the cached results.
Considerations and limitations

When using the query result reuse feature, keep in mind the following points:

- Athena reuses query results only within the same workgroup.
- Tables with row and column permissions are not supported.
- Queries that reference federated catalogs or an external Hive metastore are not supported.
- The reuse query results feature respects workgroup configurations. If you override the result configuration for a query, the feature is disabled.
- Athena requires that you have Amazon S3 read permissions for the previously generated output file to be reused.
- The reuse query results feature assumes that content of previous result has not been modified. Athena does not check the integrity of a previous result before using it.
- Potentially stale results can be returned. Athena does not check for changes in source data until the maximum reuse age that you specify has been reached.
- If multiple results are available for reuse, Athena uses the latest result.
- Queries that use non-deterministic operators or functions like `rand()` or `shuffle()` do not use cached results. For example, `LIMIT` without `ORDER` BY is non-deterministic and not cached, but `LIMIT` with `ORDER` BY is deterministic and is cached.
- You cannot reuse query results with Apache Iceberg tables.

**Reusing query results in the Athena console**

To use the feature, enable the **Reuse query results** option in the Athena query editor.
To configure the reuse query results feature

1. In the Athena query editor, under the Reuse query results option, choose the edit icon next to up to 60 minutes ago.
2. In the Edit reuse time dialog box, from the box on the right, choose a time unit (minutes, hours, or days).
3. In the box on the left, enter or choose the number of time units that you want to specify. The maximum time you can enter is the equivalent of seven days regardless of the time unit chosen.

![Edit reuse time dialog box](image)

The following example specifies a maximum reuse time of two days.

![Edit reuse time dialog box](image)

4. Choose Confirm.

A banner confirms your configuration change, and the Reuse query results option displays your new setting.
Viewing statistics and execution details for completed queries

After you run a query, you can get statistics on the input and output data processed, see a graphical representation of the time taken for each phase of the query, and interactively explore execution details.

To view query statistics for a completed query

1. After you run a query in the Athena query editor, choose the Query stats tab.
The **Query stats** tab provides the following information:

- **Data processed** – Shows you the number of input rows and bytes processed, and the number of rows and bytes output.
- **The Total runtime** – Shows the total amount of time the query took to run and a graphical representation of how much of that time was spent on queuing, planning, execution, and service processing.

**Note**
Stage-level input and output row count and data size information are not shown when a query has row-level filters defined in Lake Formation.

2. To interactively explore information about how the query ran, choose **Execution details**.
The **Execution details** page shows the execution ID for the query and a graph of the zero-based stages in the query. The stages are ordered start to finish from bottom to top. Each stage's label shows the amount of time the stage took to run.

To navigate the graph, use the following options:

- To zoom in or out, scroll the mouse, or use the magnifying icons.
- To adjust the graph to fit the screen, choose the **Zoom to fit** icon.
- To move the graph around, drag the mouse pointer.

3. To see more details for a stage, choose the stage. The stage details pane on the right shows the number of rows and bytes input and output, and an operator tree.
4. To see the stage details full width, choose the expand icon at the top right of the details pane.

5. To get information about the parts of the stage, expand one or more items in the operator tree.
For more information about execution details, see Understanding Athena EXPLAIN statement results (p. 712).

**See also**

For more information, see the following resources.

- Viewing execution plans for SQL queries (p. 340)
- Using EXPLAIN and EXPLAIN ANALYZE in Athena (p. 703)

**Working with views**

A view in Amazon Athena is a logical table, not a physical table. The query that defines a view runs each time the view is referenced in a query.
When to use views?

You may want to create views to:

- **Query a subset of data.** For example, you can create a view with a subset of columns from the original table to simplify querying data.

- **Combine multiple tables in one query.** When you have multiple tables and want to combine them with `UNION ALL`, you can create a view with that expression to simplify queries against the combined tables.

- **Hide the complexity of existing base queries and simplify queries run by users.** Base queries often include joins between tables, expressions in the column list, and other SQL syntax that make it difficult to understand and debug them. You might create a view that hides the complexity and simplifies queries.

- **Experiment with optimization techniques and create optimized queries.** For example, if you find a combination of `WHERE` conditions, `JOIN` order, or other expressions that demonstrate the best performance, you can create a view with these clauses and expressions. Applications can then make relatively simple queries against this view. If you later find a better way to optimize the original query, when you recreate the view, all the applications immediately take advantage of the optimized base query.

- **Hide the underlying table and column names, and minimize maintenance problems if those names change.** In that case, you recreate the view using the new names. All queries that use the view rather than the underlying tables keep running with no changes.

Supported actions for views in Athena

Athena supports the following actions for views. You can run these commands in the query editor.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE VIEW (p. 750)</td>
<td>Creates a new view from a specified SELECT query. For more information, see Creating views (p. 365).&lt;br&gt;The optional OR REPLACE clause lets you update the existing view by replacing it.</td>
</tr>
<tr>
<td>DESCRIBE VIEW (p. 754)</td>
<td>Shows the list of columns for the named view. This allows you to examine the attributes of a complex view.</td>
</tr>
<tr>
<td>DROP VIEW (p. 756)</td>
<td>Deletes an existing view. The optional IF EXISTS clause suppresses the error if the view does not exist.</td>
</tr>
</tbody>
</table>
Considerations for views

The following considerations apply to creating and using views in Athena:

- In Athena, you can preview and work with views created in the Athena Console, in the AWS Glue Data Catalog, if you have migrated to using it, or with Presto running on the Amazon EMR cluster connected to the same catalog. You cannot preview or add to Athena views that were created in other ways.

- If you are creating views through the AWS Glue Data Catalog, you must include the `PartitionKeys` parameter and set its value to an empty list, as follows: `"PartitionKeys":[]`. Otherwise, your view query will fail in Athena. The following example shows a view created from the Data Catalog with `"PartitionKeys":[]`:

```bash
aws glue create-table
--database-name mydb
--table-input '{
  "Name":"test",
  "TableType": "EXTERNAL_TABLE",
  "Owner": "hadoop",
  "StorageDescriptor":{
    "Columns": [{
      "Name":"a","Type":"string"},
    "InputFormat": "org.apache.hadoop.mapred.TextInputFormat",
    "OutputFormat": "org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat",
    "SerdeInfo": {"SerializationLibrary": "org.apache.hadoop.hive.serde2.OpenCSVSerde", "Parameters": {"separatorChar": ",", "serialization.format": "1"}},
    "PartitionKeys": []
  }
}
```

- If you have created Athena views in the Data Catalog, then Data Catalog treats views as tables. You can use table level fine-grained access control in Data Catalog to restrict access (p. 540) to these views.

- Athena prevents you from running recursive views and displays an error message in such cases. A recursive view is a view query that references itself.

- Athena displays an error message when it detects stale views. A stale view is reported when one of the following occurs:
  - The view references tables or databases that do not exist.
  - A schema or metadata change is made in a referenced table.
  - A referenced table is dropped and recreated with a different schema or configuration.
  - You can create and run nested views as long as the query behind the nested view is valid and the tables and databases exist.
Limitations for views

- Athena view names cannot contain special characters, other than underscore (_). For more information, see Names for tables, databases, and columns (p. 254).

- Avoid using reserved keywords for naming views. If you use reserved keywords, use double quotes to enclose reserved keywords in your queries on views. See Reserved keywords (p. 256).

- You cannot use views created in Athena with federated data sources, external Hive metastores, or UDFs. For information about working with views created externally in Hive, see Working with Hive views (p. 50).

- You cannot use views created in Athena with governed tables.

- You cannot use views with geospatial functions.

- You cannot use views to manage access control on data in Amazon S3. To query a view, you need permissions to access the data stored in Amazon S3. For more information, see Access to Amazon S3 (p. 536).

- You cannot use views with cross-account data catalogs. For information about cross-account data catalog access, see Cross-account access to AWS Glue data catalogs (p. 548).

Working with views in the console

In the Athena console, you can:

- Locate all views in the left pane, where tables are listed.

- Filter views.

- Preview a view, show its properties, edit it, or delete it.

To show the actions for a view

A view shows in the console only if you have already created it.

1. In the Athena console, choose Views, and then choose a view to expand it and show the columns in the view.

2. Choose the three vertical dots next to the view to show a list of actions for the view.
Creating views

You can create a view in the Athena console by using a template or by running an existing query.

To use a template to create a view

1. In the Athena console, next to Tables and views, choose Create, and then choose Create view.

3. Choose actions to preview the view, insert the view name into the query editor, delete the view, see the view's properties, or display and edit the view in the query editor.
Creating views

This action places an editable view template into the query editor.

2. Edit the view template according to your requirements. When you enter a name for the view in the statement, remember that view names cannot contain special characters other than underscore (_). See Names for tables, databases, and columns (p. 254). Avoid using Reserved keywords (p. 256) for naming views.

For more information about creating views, see CREATE VIEW (p. 750) and Examples of views (p. 367).

3. Choose Run to create the view. The view appears in the list of views in the Athena console.

**To create a view from an existing query**

1. Use the Athena query editor to run an existing query.
2. Under the query editor window, choose Create, and then choose View from query.
3. In the **Create View** dialog box, enter a name for the view, and then choose **Create**. View names cannot contain special characters other than underscore (_). See *Names for tables, databases, and columns* (p. 254). Avoid using *Reserved keywords* (p. 256) for naming views.

   Athena adds the view to the list of views in the console and displays the CREATE VIEW statement for the view in the query editor.

**Notes**

- If you delete a table on which a table is based and then attempt to run the view, Athena displays an error message.
- You can create a nested view, which is a view on top of an existing view. Athena prevents you from running a recursive view that references itself.

**Examples of views**

To show the syntax of the view query, use *SHOW CREATE VIEW* (p. 760).
Example Example 1

Consider the following two tables: a table employees with two columns, id and name, and a table salaries, with two columns, id and salary.

In this example, we create a view named name_salary as a SELECT query that obtains a list of IDs mapped to salaries from the tables employees and salaries:

```sql
CREATE VIEW name_salary AS
SELECT
  employees.name,
  salaries.salary
FROM employees, salaries
WHERE employees.id = salaries.id
```

Example Example 2

In the following example, we create a view named view1 that enables you to hide more complex query syntax.

This view runs on top of two tables, table1 and table2, where each table is a different SELECT query. The view selects columns from table1 and joins the results with table2. The join is based on column a that is present in both tables.

```sql
CREATE VIEW view1 AS
WITH
  table1 AS (
    SELECT a,
    MAX(b) AS the_max
    FROM x
    GROUP BY a
  ),
  table2 AS (
    SELECT a,
    AVG(d) AS the_avg
    FROM y
    GROUP BY a
  )
SELECT table1.a, table1.the_max, table2.the_avg
FROM table1
JOIN table2
ON table1.a = table2.a;
```

Using saved queries

You can use the Athena console to save, edit, run, rename, and delete the queries that you create in the query editor.

Considerations and limitations

- You can update the name, description, and query text of saved queries.
- You can only update the queries in your own account.
- You cannot change the workgroup or database to which the query belongs.
- Athena does not keep a history of query modifications. If you want to keep a particular version of a query, save it with a different name.
Working with saved queries in the Athena console

To save a query and give it a name
1. In the Athena console query editor, enter or run a query.
2. Above the query editor window, on the tab for the query, choose the three vertical dots, and then choose Save as.
3. In the Save query dialog box, enter a name for the query and an optional description. You can use the expandable Preview SQL query window to verify the contents of the query before you save it.
4. Choose Save query.

In the query editor, the tab for the query shows the name that you specified.

To run a saved query
1. In the Athena console, choose the Saved queries tab.
2. In the Saved queries list, choose the ID of the query that you want to run.

The query editor displays the query that you chose.
3. Choose Run.

To edit a saved query
1. In the Athena console, choose the Saved queries tab.
2. In the Saved queries list, choose the ID of the query that you want to edit.
3. Edit the query in the query editor.
4. Perform one of the following steps:
   • To run the query, choose Run.
   • To save the query, choose the three vertical dots on the tab for the query, and then choose Save.
   • To save the query with a different name, choose the three vertical dots on the tab for the query, and then choose Save as.

To rename or delete a saved query already displayed in the query editor
1. Choose the three vertical dots on the tab for the query, and then choose Rename or Delete.
2. Follow the prompts to rename or delete the query.

To rename a saved query not displayed in the query editor
1. In the Athena console, choose the Saved queries tab.
2. Select the check box for the query that you want to rename.
3. Choose Rename.
4. In the Rename query dialog box, edit the query name and query description. You can use the expandable Preview SQL query window to verify the contents of the query before you rename it.
5. Choose Rename query.

The renamed query appears in the Saved queries list.
To delete a saved query not displayed in the query editor

1. In the Athena console, choose the Saved queries tab.
2. Select one or more check boxes for the queries that you want to delete.
3. Choose Delete.
4. At the confirmation prompt, choose Delete.

   One or more queries are removed from the Saved queries list.

Using the Athena API to update saved queries

For information about using the Athena API to update a saved query, see the UpdateNamedQuery action in the Athena API Reference.

Using parameterized queries

You can use Athena parameterized queries to re-run the same query with different parameter values at execution time and help prevent SQL injection attacks. In Athena, parameterized queries can take the form of execution parameters in any DML query or SQL prepared statements.

- Queries with execution parameters can be done in a single step and are not workgroup specific. You place question marks in any DML query for the values that you want to parameterize. When you run the query, you declare the execution parameter values sequentially. The declaration of parameters and the assigning of values for the parameters can be done in the same query, but in a decoupled fashion. Unlike prepared statements, you can select the workgroup when you submit a query with execution parameters.
- Prepared statements require two separate SQL statements: PREPARE and EXECUTE. First, you define the parameters in the PREPARE statement. Then, you run an EXECUTE statement that supplies the values for the parameters that you defined. Prepared statements are workgroup specific; you cannot run them outside the context of the workgroup to which they belong.

Considerations and limitations

- Parameterized queries are supported only in Athena engine version 2 and later versions. For information about Athena engine versions, see Athena engine versioning (p. 669).
- Currently, parameterized queries are supported only for SELECT, INSERT INTO, CTAS, and UNLOAD statements.
- In parameterized queries, parameters are positional and are denoted by ?. Parameters are assigned values by their order in the query. Named parameters are not supported.
- Currently, ? parameters can be placed only in the WHERE clause. Syntax like SELECT ? FROM table is not supported.
- Question mark parameters cannot be placed in double or single quotes (that is, '?' and "?" are not valid syntax).
- Prepared statements are workgroup specific, and prepared statement names must be unique within the workgroup.
- IAM permissions for prepared statements are required. For more information, see Allow access to prepared statements (p. 554).
- Queries with execution parameters in the Athena console are limited to a maximum of 25 question marks.
Querying using execution parameters

You can use question mark placeholders in any DML query to create a parameterized query without creating a prepared statement first. To run these queries, you can use the Athena console, or use the AWS CLI or the AWS SDK and declare the variables in the execution-parameters argument.

Running queries with execution parameters in the Athena console

When you run a parameterized query that has execution parameters (question marks) in the Athena console, you are prompted for the values in the order in which the question marks occur in the query.

To run a query that has execution parameters

1. Enter a query with question mark placeholders in the Athena editor, as in the following example.

   ```sql
   SELECT * FROM "my_database"."my_table"
   WHERE year = ? and month= ? and day= ?
   ```

2. Choose Run.
3. In the Enter parameters dialog box, enter a value in order for each of the question marks in the query.

4. When you are finished entering the parameters, choose Run. The editor shows the query results for the parameter values that you entered.

At this point, you can do one of the following:

- Enter different parameter values for the same query, and then choose Run again.
- To clear all of the values that you entered at once, choose Clear.
- To edit the query directly (for example, to add or remove question marks), close the Enter parameters dialog box first.
To save the parameterized query for later use, choose Save or Save as, and then give the query a name. For more information about using saved queries, see Using saved queries (p. 368).

As a convenience, the Enter parameters dialog box remembers the values that you entered previously for the query as long as you use the same tab in the query editor.

Running queries with execution parameters using the AWS CLI

To use the AWS CLI to run queries with execution parameters, use the start-query-execution command and provide a parameterized query in the query-string argument. Then, in the execution-parameters argument, provide the values for the execution parameters. The following example illustrates this technique.

```
aws athena start-query-execution
   --query-string "SELECT * FROM table WHERE x = ? AND y = ?"
   --query-execution-context "Database"="default"
   --result-configuration "OutputLocation"="s3://..."
   --execution-parameters "1" "2"
```

Querying with prepared statements

You can use a prepared statement for repeated execution of the same query with different query parameters. A prepared statement contains parameter placeholders whose values are supplied at execution time.

**Note**

*The maximum number of prepared statements in a workgroup is 1000.*

SQL statements

You can use the PREPARE, EXECUTE and DEALLOCATE PREPARE SQL statements to run parameterized queries in the Athena console query editor.

- To specify parameters where you would normally use literal values, use question marks in the PREPARE statement.
- To replace the parameters with values when you run the query, use the USING clause in the EXECUTE statement.
- To remove a prepared statement from the prepared statements in a workgroup, use the DEALLOCATE PREPARE statement.

The following sections provide additional detail about each of these statements.

**PREPARE**

Prepares a statement to be run at a later time. Prepared statements are saved in the current workgroup with the name that you specify. The statement can include parameters in place of literals to be replaced when the query is run. Parameters to be replaced by values are denoted by question marks.

**Syntax**

```
PREPARE statement_name FROM statement
```
The following table describes these parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>statement_name</em></td>
<td>The name of the statement to be prepared. The name must be unique within the workgroup.</td>
</tr>
<tr>
<td>statement</td>
<td>A SELECT, CTAS, or INSERT INTO query.</td>
</tr>
</tbody>
</table>

**PREPARE examples**

The following examples show the use of the PREPARE statement. Question marks denote the values to be supplied by the EXECUTE statement when the query is run.

```sql
PREPARE my_select1 FROM
    SELECT * FROM nation
```

```sql
PREPARE my_select2 FROM
    SELECT * FROM "my_database"."my_table" WHERE year = ?
```

```sql
PREPARE my_select3 FROM
    SELECT order FROM orders WHERE productid = ? and quantity < ?
```

```sql
PREPARE my_insert FROM
    INSERT INTO cities_usa (city, state)
    SELECT city, state
    FROM cities_world
    WHERE country = ?
```

```sql
PREPARE my_unload FROM
    UNLOAD (SELECT * FROM table1 WHERE productid < ?)
    TO 's3://my_output_bucket/'
    WITH (format='PARQUET')
```

**EXECUTE**

Runs a prepared statement. Values for parameters are specified in the USING clause.

**Syntax**

```sql
EXECUTE statement_name [USING value1 [ ,value2, ... ] ]
```

*statement_name* is the name of the prepared statement. *value1* and *value2* are the values to be specified for the parameters in the statement.

**EXECUTE examples**

The following example runs the my_select1 prepared statement, which contains no parameters.

```sql
EXECUTE my_select1
```
The following example runs the `my_select2` prepared statement, which contains a single parameter.

```
EXECUTE my_select2 USING 2012
```

The following example runs the `my_select3` prepared statement, which has two parameters.

```
EXECUTE my_select3 USING 346078, 12
```

The following example supplies a string value for a parameter in the prepared statement `my_insert`.

```
EXECUTE my_insert USING 'usa'
```

The following example supplies a numerical value for the `productid` parameter in the prepared statement `my_unload`.

```
EXECUTE my_unload USING 12
```

**DEALLOCATE PREPARE**

Removes the prepared statement with the specified name from the list of prepared statements in the current workgroup.

**Syntax**

```
DEALLOCATE PREPARE statement_name
```

`statement_name` is the name of the prepared statement to be removed.

**Example**

The following example removes the `my_select1` prepared statement from the current workgroup.

```
DEALLOCATE PREPARE my_select1
```

**Executing prepared statements without the USING clause in the Athena console**

If you run an existing prepared statement with the syntax `EXECUTE prepared_statement` in the query editor, Athena opens the Enter parameters dialog box so that you can enter the values that would normally go in the USING clause of the `EXECUTE ... USING` statement.

**To run a prepared statement using the Enter parameters dialog box**

1. In the query editor, instead of using the syntax `EXECUTE prepared_statement USING value1, value2 ...`, use the syntax `EXECUTE prepared_statement`.
2. Choose Run. The Enter parameters dialog box appears.
3. Enter the values in order in the **Execution parameters** dialog box. Because the original text of the query is not visible, you must remember the meaning of each positional parameter or have the prepared statement available for reference.

4. Choose **Run**.

### Creating prepared statements using the AWS CLI

To use the AWS CLI to create a prepared statement, you can use one of the following *athena* commands:

- Use the `create-prepared-statement` command and provide a query statement that has execution parameters.
- Use the `start-query-execution` command and provide a query string that uses the `PREPARE` syntax.

#### Using create-prepared-statement

In a `create-prepared-statement` command, define the query text in the `query-statement` argument, as in the following example:

```
aws athena create-prepared-statement
--statement-name PreparedStatement1
--query-statement "SELECT * FROM table WHERE x = ?"
--work-group athena-engine-v2
```

#### Using start-query-execution and the `PREPARE` syntax

Use the `start-query-execution` command. Put the `PREPARE` statement in the `query-string` argument, as in the following example:

```
aws athena start-query-execution
```
Executing prepared statements using the AWS CLI

To execute a prepared statement with the AWS CLI, you can supply values for the parameters by using one of the following methods:

- Use the `execution-parameters` argument.
- Use the `EXECUTE ... USING` SQL syntax in the `query-string` argument.

Using the `execution-parameters` argument

In this approach, you use the `start-query-execution` command and provide the name of an existing prepared statement in the `query-string` argument. Then, in the `execution-parameters` argument, you provide the values for the execution parameters. The following example shows this method.

```
aws athena start-query-execution
--query-string "Execute PreparedStatement1"
--query-execution-context '{"Database": "default"}'
--result-configuration '{"OutputLocation": "s3://..."}'
--execution-parameters "1" "2"
```

Using the `EXECUTE ... USING` SQL syntax

To run an existing prepared statement using the `EXECUTE ... USING` syntax, you use the `start-query-execution` command and place the both the name of the prepared statement and the parameter values in the `query-string` argument, as in the following example:

```
aws athena start-query-execution
--query-string "EXECUTE PreparedStatement1 USING 1"
--query-execution-context '{"Database": "default"}'
--result-configuration '{"OutputLocation": "s3://..."}'
```

See also

See the following related posts in the AWS Big Data Blog.

- Improve reusability and security using Amazon Athena parameterized queries
- Use Amazon Athena parameterized queries to provide data as a service

Handling schema updates

This section provides guidance on handling schema updates for various data formats. Athena is a schema-on-read query engine. This means that when you create a table in Athena, it applies schemas when reading the data. It does not change or rewrite the underlying data.

If you anticipate changes in table schemas, consider creating them in a data format that is suitable for your needs. Your goals are to reuse existing Athena queries against evolving schemas, and avoid schema mismatch errors when querying tables with partitions.

To achieve these goals, choose a table’s data format based on the table in the following topic.
Summary: Updates and data formats in Athena

The following table summarizes data storage formats and their supported schema manipulations. Use this table to help you choose the format that will enable you to continue using Athena queries even as your schemas change over time.

In this table, observe that Parquet and ORC are columnar formats with different default column access methods. By default, Parquet will access columns by name and ORC by index (ordinal value). Therefore, Athena provides a SerDe property defined when creating a table to toggle the default column access method which enables greater flexibility with schema evolution.

For Parquet, the `parquet.column.index.access` property may be set to `true`, which sets the column access method to use the column's ordinal number. Setting this property to `false` will change the column access method to use column name. Similarly, for ORC use the `orc.column.index.access` property to control the column access method. For more information, see Index access in ORC and parquet (p. 378).

CSV and TSV allow you to do all schema manipulations except reordering of columns, or adding columns at the beginning of the table. For example, if your schema evolution requires only renaming columns but not removing them, you can choose to create your tables in CSV or TSV. If you require removing columns, do not use CSV and TSV, and instead use any of the other supported formats, preferably, a columnar format, such as Parquet or ORC.

<table>
<thead>
<tr>
<th>Expected type of schema update</th>
<th>Summary</th>
<th>CSV (with and without headers) and TSV</th>
<th>JSON</th>
<th>AVRO</th>
<th>PARQUET Read by name (default)</th>
<th>PARQUET Read by index</th>
<th>ORC: Read by index (default)</th>
<th>ORC: Read by name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename columns (p. 382)</td>
<td>Store your data in CSV and TSV, or in ORC and Parquet if they are read by index.</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Add columns at the beginning or in the middle of the table (p. 380)</td>
<td>Store your data in JSON, AVRO, or in Parquet and ORC if they are read by name. Do not use CSV and TSV.</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Add columns at the end of the table (p. 381)</td>
<td>Store your data in CSV or TSV, JSON, AVRO, ORC, or Parquet.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
### Expected type of schema update

<table>
<thead>
<tr>
<th>Summary</th>
<th>CSV (with and without headers) and TSV</th>
<th>JSON</th>
<th>AVRO</th>
<th>PARQUET Read by name (default)</th>
<th>PARQUET Read by index (default)</th>
<th>ORC: Read by name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove columns (p. 381)</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Store your data in CSV, AVRO, or Parquet and ORC, if they are read by name. Do not use CSV and TSV.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reorder columns (p. 383)</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Store your data in AVRO, JSON or ORC and Parquet if they are read by name.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Change a column's data type (p. 383)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Store your data in any format, but test your query in Athena to make sure the data types are compatible. For Parquet and ORC, changing a data type works only for partitioned tables.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Index access in ORC and parquet

PARQUET and ORC are columnar data storage formats that can be read by index, or by name. Storing your data in either of these formats lets you perform all operations on schemas and run Athena queries without schema mismatch errors.

- Athena reads **ORC by index by default**, as defined in SERDEPROPERTIES (`'orc.column.index.access'='true'`). For more information, see **ORC: Read by index** (p. 379).

- Athena reads **Parquet by name by default**, as defined in SERDEPROPERTIES (`'parquet.column.index.access'='false'`). For more information, see **Parquet: Read by name** (p. 379).

Since these are defaults, specifying these SerDe properties in your CREATE TABLE queries is optional, they are used implicitly. When used, they allow you to run some schema update operations while preventing other such operations. To enable those operations, run another CREATE TABLE query and change the SerDe settings.

**Note**

The SerDe properties are not automatically propagated to each partition. Use ALTER TABLE ADD PARTITION statements to set the SerDe properties for each partition. To automate this process, write a script that runs ALTER TABLE ADD PARTITION statements.

The following sections describe these cases in detail.
ORC: Read by index

A table in ORC is read by index, by default. This is defined by the following syntax:

```sql
WITH SERDEPROPERTIES (  
'orc.column.index.access'='true')
```

Reading by index allows you to rename columns. But then you lose the ability to remove columns or add them in the middle of the table.

To make ORC read by name, which will allow you to add columns in the middle of the table or remove columns in ORC, set the SerDe property `orc.column.index.access` to `false` in the `CREATE TABLE` statement. In this configuration, you will lose the ability to rename columns.

**Note**

When ORC tables are set to read by name, Athena requires that all column names in the ORC files be in lower case. Because Apache Spark does not lowercase field names when it generates ORC files, Athena might not be able to read the data so generated. The workaround is to rename the columns to be in lower case.

The following example illustrates how to change the ORC to make it read by name:

```sql
CREATE EXTERNAL TABLE orders_orc_read_by_name (  
'o_comment' string,  
'o_orderkey' int,  
'o_custkey' int,  
'o_orderpriority' string,  
'o_orderstatus' string,  
'o_clerk' string,  
'o_shippriority' int,  
'o_orderdate' string)  
ROW FORMAT SERDE  
'org.apache.hadoop.hive.ql.io.orc.OrcSerde'  
WITH SERDEPROPERTIES (  
'orc.column.index.access'='false')  
STORED AS INPUTFORMAT  
'org.apache.hadoop.hive.ql.io.orc.OrcInputFormat'  
OUTPUTFORMAT  
'org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat'  
LOCATION 's3://schema_updates/orders_orc/';
```

Parquet: Read by name

A table in Parquet is read by name, by default. This is defined by the following syntax:

```sql
WITH SERDEPROPERTIES (  
'parquet.column.index.access'='true')
```

Reading by name allows you to add columns in the middle of the table and remove columns. But then you lose the ability to rename columns.

To make Parquet read by index, which will allow you to rename columns, you must create a table with `parquet.column.index.access` SerDe property set to `true`.

Types of updates

Here are the types of updates that a table's schema can have. We review each type of schema update and specify which data formats allow you to have them in Athena.
• Adding columns at the beginning or in the middle of the table (p. 380)
• Adding columns at the end of the table (p. 381)
• Removing columns (p. 381)
• Renaming columns (p. 382)
• Reordering columns (p. 383)
• Changing a column’s data type (p. 383)

Depending on how you expect your schemas to evolve, to continue using Athena queries, choose a compatible data format.

Let’s consider an application that reads orders information from an orders table that exists in two formats: CSV and Parquet.

The following example creates a table in Parquet:

```sql
CREATE EXTERNAL TABLE orders_parquet (  
  `orderkey` int,
  `orderstatus` string,
  `totalprice` double,
  `orderdate` string,
  `orderpriority` string,
  `clerk` string,
  `shippriority` int
) STORED AS PARQUET
LOCATION 's3://schema_updates/orders_parquet/';
```

The following example creates the same table in CSV:

```sql
CREATE EXTERNAL TABLE orders_csv (  
  `orderkey` int,
  `orderstatus` string,
  `totalprice` double,
  `orderdate` string,
  `orderpriority` string,
  `clerk` string,
  `shippriority` int
) ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION 's3://schema_updates/orders_csv/';
```

In the following sections, we review how updates to these tables affect Athena queries.

### Adding columns at the beginning or in the middle of the table

Adding columns is one of the most frequent schema changes. For example, you may add a new column to enrich the table with new data. Or, you may add a new column if the source for an existing column has changed, and keep the previous version of this column, to adjust applications that depend on them.

To add columns at the beginning or in the middle of the table, and continue running queries against existing tables, use AVRO, JSON, and Parquet and ORC if their SerDe property is set to read by name. For information, see [Index access in ORC and parquet](p. 378).

Do not add columns at the beginning or in the middle of the table in CSV and TSV, as these formats depend on ordering. Adding a column in such cases will lead to schema mismatch errors when the schema of partitions changes.

The following example shows adding a column to a JSON table in the middle of the table:
Types of updates

CREATE EXTERNAL TABLE orders_json_column_addition (  
  'o_orderkey' int,  
  'o_custkey' int,  
  'o_orderstatus' string,  
  'o_comment' string,  
  'o_totalprice' double,  
  'o_orderdate' string,  
  'o_orderpriority' string,  
  'o_clerk' string,  
  'o_shippriority' int,  
)  
ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'  
LOCATION 's3://schema_updates/orders_json/';

Adding columns at the end of the table

If you create tables in any of the formats that Athena supports, such as Parquet, ORC, Avro, JSON, CSV, and TSV, you can use the ALTER TABLE ADD COLUMNS statement to add columns after existing columns but before partition columns.

The following example adds a comment column at the end of the orders_parquet table before any partition columns:

ALTER TABLE orders_parquet ADD COLUMNS (comment string)

Note
To see a new table column in the Athena Query Editor after you run ALTER TABLE ADD COLUMNS, manually refresh the table list in the editor, and then expand the table again.

Removing columns

You may need to remove columns from tables if they no longer contain data, or to restrict access to the data in them.

• You can remove columns from tables in JSON, Avro, and in Parquet and ORC if they are read by name. For information, see Index access in ORC and parquet (p. 378).

• We do not recommend removing columns from tables in CSV and TSV if you want to retain the tables you have already created in Athena. Removing a column breaks the schema and requires that you recreate the table without the removed column.

In this example, remove a column `totalprice` from a table in Parquet and run a query. In Athena, Parquet is read by name by default, this is why we omit the SERDEPROPERTIES configuration that specifies reading by name. Notice that the following query succeeds, even though you changed the schema:

CREATE EXTERNAL TABLE orders_parquet_column_removed (  
  'o_orderkey' int,  
  'o_custkey' int,  
  'o_orderstatus' string,  
  'o_orderdate' string,  
  'o_orderpriority' string,  
  'o_clerk' string,  
  'o_shippriority' int,  
  'o_comment' string  
)  
STORED AS PARQUET  
LOCATION 's3://schema_updates/orders_parquet/';
Renaming columns

You may want to rename columns in your tables to correct spelling, make column names more descriptive, or to reuse an existing column to avoid column reordering.

You can rename columns if you store your data in CSV and TSV, or in Parquet and ORC that are configured to read by index. For information, see Index access in ORC and parquet (p. 378).

Athena reads data in CSV and TSV in the order of the columns in the schema and returns them in the same order. It does not use column names for mapping data to a column, which is why you can rename columns in CSV or TSV without breaking Athena queries.

One strategy for renaming columns is to create a new table based on the same underlying data, but using new column names. The following example creates a new orders_parquet table called orders_parquet_column_renamed. The example changes the column `o_totalprice` name to `o_total_price` and then runs a query in Athena:

```sql
CREATE EXTERNAL TABLE orders_parquet_column_renamed (  'o_orderkey' int,  'o_custkey' int,  'o_orderstatus' string,  'o_total_price' double,  'o_orderdate' string,  'o_orderpriority' string,  'o_clerk' string,  'o_shippriority' int,  'o_comment' string)
STORED AS PARQUET
LOCATION 's3://schema_updates/orders_parquet/';
```

In the Parquet table case, the following query runs, but the renamed column does not show data because the column was being accessed by name (a default in Parquet) rather than by index:

```sql
SELECT *
FROM orders_parquet_column_renamed;
```

A query with a table in CSV looks similar:

```sql
CREATE EXTERNAL TABLE orders_csv_column_renamed (  'o_orderkey' int,  'o_custkey' int,  'o_orderstatus' string,  'o_total_price' double,  'o_orderdate' string,  'o_orderpriority' string,  'o_clerk' string,  'o_shippriority' int,  'o_comment' string)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION 's3://schema_updates/orders_csv/';
```

In the CSV table case, the following query runs and the data displays in all columns, including the one that was renamed:

```sql
SELECT *
FROM orders_csv_column_renamed;
```
Reordering columns

You can reorder columns only for tables with data in formats that read by name, such as JSON or Parquet, which reads by name by default. You can also make ORC read by name, if needed. For information, see Index access in ORC and parquet (p. 378).

The following example illustrates reordering of columns:

```sql
CREATE EXTERNAL TABLE orders_parquet_columns_reordered (
    'o_comment' string,
    'o_orderkey' int,
    'o_custkey' int,
    'o_orderpriority' string,
    'o_orderstatus' string,
    'o_clerk' string,
    'o_shippriority' int,
    'o_orderdate' string
) STORED AS PARQUET
LOCATION 's3://schema_updates/orders_parquet/';
```

Changing a column's data type

You change column types because a column's data type can no longer hold the amount of information, for example, when an ID column exceeds the size of an INT data type and has to change to a BIGINT data type.

Changing a column's data type has these limitations:

- Only certain data types can be converted to other data types. See the table in this section for data types that can change.
- For data in Parquet and ORC, you cannot change a column's data type if the table is not partitioned.

For partitioned tables in Parquet and ORC, a partition's column type can be different from another partition's column type, and Athena will CAST to the desired type, if possible. For information, see Avoiding schema mismatch errors for tables with partitions (p. 384).

**Important**

We strongly suggest that you test and verify your queries before performing data type translations. If Athena cannot convert the data type from the original data type to the target data type, the CREATE TABLE query may fail.

The following table lists data types that you can change:

<table>
<thead>
<tr>
<th>Original data type</th>
<th>Available target data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING</td>
<td>BYTE, TINYINT, SMALLINT, INT, BIGINT</td>
</tr>
<tr>
<td>BYTE</td>
<td>TINYINT, SMALLINT, INT, BIGINT</td>
</tr>
<tr>
<td>TINYINT</td>
<td>SMALLINT, INT, BIGINT</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>INT, BIGINT</td>
</tr>
<tr>
<td>INT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>FLOAT</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>
In the following example of the `orders_json` table, change the data type for the column `o_shippriority` to BIGINT:

```sql
CREATE EXTERNAL TABLE orders_json (  
  'o_orderkey' int,
  'o_custkey' int,
  'o_orderstatus' string,
  'o_totalprice' double,
  'o_orderdate' string,
  'o_orderpriority' string,
  'o_clerk' string,
  'o_shippriority' BIGINT
)
ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'
LOCATION 's3://schema_updates/orders_json';
```

The following query runs successfully, similar to the original SELECT query, before the data type change:

```sql
SELECT * FROM orders_json
LIMIT 10;
```

### Updates in tables with partitions

In Athena, a table and its partitions must use the same data formats but their schemas may differ. When you create a new partition, that partition usually inherits the schema of the table. Over time, the schemas may start to differ. Reasons include:

- If your table's schema changes, the schemas for partitions are not updated to remain in sync with the table's schema.
- The AWS Glue Crawler allows you to discover data in partitions with different schemas. This means that if you create a table in Athena with AWS Glue, after the crawler finishes processing, the schemas for the table and its partitions may be different.
- If you add partitions directly using an AWS API.

Athena processes tables with partitions successfully if they meet the following constraints. If these constraints are not met, Athena issues a HIVE_PARTITION_SCHEMA_MISMATCH error.

- Each partition's schema is compatible with the table's schema.
- The table's data format allows the type of update you want to perform: add, delete, reorder columns, or change a column's data type.

For example, for CSV and TSV formats, you can rename columns, add new columns at the end of the table, and change a column's data type if the types are compatible, but you cannot remove columns. For other formats, you can add or remove columns, or change a column's data type to another if the types are compatible. For information, see [Summary: Updates and Data Formats in Athena](p. 377).

### Avoiding schema mismatch errors for tables with partitions

At the beginning of query execution, Athena verifies the table's schema by checking that each column data type is compatible between the table and the partition.

- For Parquet and ORC data storage types, Athena relies on the column names and uses them for its column name-based schema verification. This eliminates HIVE_PARTITION_SCHEMA_MISMATCH errors for tables with partitions in Parquet and ORC. (This is true for ORC if the SerDe property is set to
access the index by name: `orc.column.index.access=FALSE`. Parquet reads the index by name by default).

- For CSV, JSON, and Avro, Athena uses an index-based schema verification. This means that if you encounter a schema mismatch error, you should drop the partition that is causing a schema mismatch and recreate it, so that Athena can query it without failing.

Athena compares the table's schema to the partition schemas. If you create a table in CSV, JSON, and AVRO in Athena with AWS Glue Crawler, after the Crawler finishes processing, the schemas for the table and its partitions may be different. If there is a mismatch between the table's schema and the partition schemas, your queries fail in Athena due to the schema verification error similar to this: 'crawler_test.click_avro' is declared as type 'string', but partition 'partition_0=2017-01-17' declared column 'col68' as type 'double'.

A typical workaround for such errors is to drop the partition that is causing the error and recreate it. For more information, see ALTER TABLE DROP PARTITION (p. 735) and ALTER TABLE ADD PARTITION (p. 733).

## Querying arrays

Amazon Athena lets you create arrays, concatenate them, convert them to different data types, and then filter, flatten, and sort them.

### Topics
- Creating arrays (p. 385)
- Concatenating strings and arrays (p. 387)
- Converting array data types (p. 388)
- Finding lengths (p. 388)
- Accessing array elements (p. 389)
- Flattening nested arrays (p. 390)
- Creating arrays from subqueries (p. 392)
- Filtering arrays (p. 393)
- Sorting arrays (p. 394)
- Using aggregation functions with arrays (p. 394)
- Converting arrays to strings (p. 395)
- Using arrays to create maps (p. 396)
- Querying arrays with complex types and nested structures (p. 397)

### Creating arrays

To build an array literal in Athena, use the `ARRAY` keyword, followed by brackets `[]`, and include the array elements separated by commas.

### Examples

This query creates one array with four elements.

```sql
SELECT ARRAY [1,2,3,4] AS items
```

It returns:
Creating arrays

This query creates two arrays.

```
SELECT ARRAY[ ARRAY[1,2], ARRAY[3,4] ] AS items
```

It returns:

```
+--------------------+
| items              |
+--------------------+
| [[1, 2], [3, 4]]   |
+--------------------+
```

To create an array from selected columns of compatible types, use a query, as in this example:

```
WITH dataset AS (  
    SELECT 1 AS x, 2 AS y, 3 AS z  
)  
SELECT ARRAY [x,y,z] AS items FROM dataset
```

This query returns:

```
+-----------+  
| items     |  
+-----------+  
| [1,2,3]   |  
+-----------+  
```

In the following example, two arrays are selected and returned as a welcome message.

```
WITH dataset AS (  
    SELECT  
        ARRAY ['hello', 'amazon', 'athena'] AS words,  
        ARRAY ['hi', 'alexa'] AS alexa  
)  
SELECT ARRAY[words, alexa] AS welcome_msg FROM dataset
```

This query returns:

```
+----------------------------------------+  
| welcome_msg                            |  
+----------------------------------------+  
| [[hello, amazon, athena], [hi, alexa]] |  
+----------------------------------------+  
```

To create an array of key-value pairs, use the MAP operator that takes an array of keys followed by an array of values, as in this example:

```
SELECT ARRAY[
```
MAP(ARRAY['first', 'last', 'age'],ARRAY['Bob', 'Smith', '40']),
MAP(ARRAY['first', 'last', 'age'],ARRAY['Jane', 'Doe', '30']),
MAP(ARRAY['first', 'last', 'age'],ARRAY['Billy', 'Smith', '8'])
] AS people

This query returns:

<table>
<thead>
<tr>
<th>people</th>
</tr>
</thead>
<tbody>
<tr>
<td>[{last=Smith, first=Bob, age=40}, {last=Doe, first=Jane, age=30}, {last=Smith, first=Billy, age=8}]</td>
</tr>
</tbody>
</table>

Concatenating strings and arrays

Concatenating strings

To concatenate two strings, you can use the double pipe || operator, as in the following example.

SELECT 'This' || ' is' || ' a' || ' test.' AS Concatenated_String

This query returns:

Concatenated_String
This is a test.

You can use the concat() function to achieve the same result.

SELECT concat('This', ' is', ' a', ' test.') AS Concatenated_String

This query returns:

Concatenated_String
This is a test.

Concatenating arrays

You can use the same techniques to concatenate arrays.

To concatenate multiple arrays, use the double pipe || operator.

SELECT ARRAY [4,5] || ARRAY[ ARRAY[1,2], ARRAY[3,4] ] AS items

This query returns:

items
[[4, 5], [1, 2], [3, 4]]
To combine multiple arrays into a single array, use the double pipe operator or the `concat()` function.

WITH
dataset AS (
    SELECT
        ARRAY ['Hello', 'Amazon', 'Athena'] AS words,
        ARRAY ['Hi', 'Alexa'] AS alexa
    
    SELECT concat(words, alexa) AS welcome_msg
FROM dataset

This query returns:

<table>
<thead>
<tr>
<th>welcome_msg</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Hello, Amazon, Athena, Hi, Alexa]</td>
</tr>
</tbody>
</table>

For more information about `concat()` other string functions, see String functions and operators in the Trino documentation.

### Converting array data types

To convert data in arrays to supported data types, use the CAST operator, as `CAST(value AS type)`. Athena supports all of the native Presto data types.

SELECT
    ARRAY [CAST(4 AS VARCHAR), CAST(5 AS VARCHAR)]
AS items

This query returns:

| +-------+ |
|---------+|
| | items | |
| +-------+ |
| | [4,5]  | |

Create two arrays with key-value pair elements, convert them to JSON, and concatenate, as in this example:

SELECT
    ARRAY[CAST(MAP(ARRAY['a1', 'a2', 'a3'], ARRAY[1, 2, 3]) AS JSON)] ||
    ARRAY[CAST(MAP(ARRAY['b1', 'b2', 'b3'], ARRAY[4, 5, 6]) AS JSON)]
AS items

This query returns:

| +--------------------------------------------------+ |
| | items                                            | |
| +--------------------------------------------------+ |
| | ["a1":1,"a2":2,"a3":3],{"b1":4,"b2":5,"b3":6}] | |

### Finding lengths

The `cardinality` function returns the length of an array, as in this example:
Amazon Athena User Guide
Accessing array elements

SELECT cardinality(ARRAY[1,2,3,4]) AS item_count

This query returns:
+------------+
| item_count |
+------------+
| 4
|
+------------+

Accessing array elements
To access array elements, use the [] operator, with 1 specifying the ﬁrst element, 2 specifying the
second element, and so on, as in this example:
WITH dataset AS (
SELECT
ARRAY[CAST(MAP(ARRAY['a1', 'a2', 'a3'], ARRAY[1, 2, 3]) AS JSON)] ||
ARRAY[CAST(MAP(ARRAY['b1', 'b2', 'b3'], ARRAY[4, 5, 6]) AS JSON)]
AS items )
SELECT items[1] AS item FROM dataset

This query returns:
+------------------------+
| item
|
+------------------------+
| {"a1":1,"a2":2,"a3":3} |
+------------------------+

To access the elements of an array at a given position (known as the index position), use the
element_at() function and specify the array name and the index position:
• If the index is greater than 0, element_at() returns the element that you specify, counting from the
beginning to the end of the array. It behaves as the [] operator.
• If the index is less than 0, element_at() returns the element counting from the end to the beginning
of the array.
The following query creates an array words, and selects the ﬁrst element hello from it as the
first_word, the second element amazon (counting from the end of the array) as the middle_word,
and the third element athena, as the last_word.
WITH dataset AS (
SELECT ARRAY ['hello', 'amazon', 'athena'] AS words
)
SELECT
element_at(words, 1) AS first_word,
element_at(words, -2) AS middle_word,
element_at(words, cardinality(words)) AS last_word
FROM dataset

This query returns:
+----------------------------------------+
| first_word | middle_word | last_word |

389


Flattening nested arrays

When working with nested arrays, you often need to expand nested array elements into a single array, or expand the array into multiple rows.

Examples

To flatten a nested array's elements into a single array of values, use the `flatten` function. This query returns a row for each element in the array.

```sql
SELECT flatten(ARRAY[ ARRAY[1,2], ARRAY[3,4] ]) AS items
```

This query returns:

```
+-----------+
| items     |
+-----------+
| [1,2,3,4] |
+-----------+
```

To flatten an array into multiple rows, use `CROSS JOIN` in conjunction with the `UNNEST` operator, as in this example:

```sql
WITH dataset AS (
    SELECT
        'engineering' as department,
        ARRAY['Sharon', 'John', 'Bob', 'Sally'] as users
    )
SELECT department, names FROM dataset
CROSS JOIN UNNEST(users) as t(names)
```

This query returns:

```
+----------------------+
| department | names  |
+----------------------+
| engineering | Sharon |
+----------------------|
| engineering | John   |
+----------------------|
| engineering | Bob    |
+----------------------|
| engineering | Sally  |
+----------------------+
```

To flatten an array of key-value pairs, transpose selected keys into columns, as in this example:

```sql
WITH dataset AS (
    SELECT
        'engineering' as department,
        ARRAY[
            MAP(ARRAY['first', 'last', 'age'], ARRAY['Bob', 'Smith', '40']),
        ]
    )
SELECT department, first, last, age FROM dataset
```

This query returns:

```
+----------------------+
| department | first  | last  | age   |
+----------------------+
| engineering | Bob    | Smith | 40    |
+----------------------+
```
MAP(ARRAY['first', 'last', 'age'], ARRAY['Jane', 'Doe', '30']),
MAP(ARRAY['first', 'last', 'age'], ARRAY['Billy', 'Smith', '8'])] AS people

This query returns:

<table>
<thead>
<tr>
<th>first_name</th>
<th>last_name</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>Smith</td>
<td>engineering</td>
</tr>
<tr>
<td>Jane</td>
<td>Doe</td>
<td>engineering</td>
</tr>
<tr>
<td>Billy</td>
<td>Smith</td>
<td>engineering</td>
</tr>
</tbody>
</table>

From a list of employees, select the employee with the highest combined scores. UNNEST can be used in the FROM clause without a preceding CROSS JOIN as it is the default join operator and therefore implied.

WITH
dataset AS (
  SELECT ARRAY[
    CAST(ROW('Sally', 'engineering', ARRAY[1,2,3,4]) AS ROW(name VARCHAR, department VARCHAR, scores ARRAY(INTEGER))],
    CAST(ROW('John', 'finance', ARRAY[7,8,9]) AS ROW(name VARCHAR, department VARCHAR, scores ARRAY(INTEGER))],
    CAST(ROW('Amy', 'devops', ARRAY[12,13,14,15]) AS ROW(name VARCHAR, department VARCHAR, scores ARRAY(INTEGER))]
] AS users
),
users AS (
  SELECT person, score
  FROM dataset,
  UNNEST(dataset.users) AS t(person),
  UNNEST(person.scores) AS t(score)
)
SELECT person.name, person.department, SUM(score) AS total_score FROM users
GROUP BY (person.name, person.department)
ORDER BY (total_score) DESC
LIMIT 1

This query returns:

<table>
<thead>
<tr>
<th>name</th>
<th>department</th>
<th>total_score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amy</td>
<td>devops</td>
<td>54</td>
</tr>
</tbody>
</table>

From a list of employees, select the employee with the highest individual score.

WITH
dataset AS (
  SELECT ARRAY[...]
] AS people

This query returns:

<table>
<thead>
<tr>
<th>name</th>
<th>department</th>
<th>total_score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amy</td>
<td>devops</td>
<td>54</td>
</tr>
</tbody>
</table>

From a list of employees, select the employee with the highest individual score.
Creating arrays from subqueries

Create an array from a collection of rows.

WITH dataset AS (  
  SELECT ARRAY[1,2,3,4,5] AS items  
)  
SELECT array_agg(i) AS array_items  
FROM dataset  
CROSS JOIN UNNEST(items) AS t(i)

This query returns:

+-----------------+
| array_items     |
+-----------------+
| [1, 2, 3, 4, 5] |
+-----------------+

To create an array of unique values from a set of rows, use the distinct keyword.

WITH

Considerations and limitations

- CROSS JOIN does not generate a Cartesian product if the main (FROM) table has an empty array or NULL on the specified column.
- Currently, UNNEST can be used only with CROSS JOIN and does not support other JOIN types (for example, LEFT JOIN UNNEST).
Filtering arrays

Create an array from a collection of rows if they match the filter criteria.

WITH
dataset AS (  
  SELECT ARRAY [1,2,3,4,5] AS items  
)  
SELECT array_agg(i) AS array_items  
FROM dataset  
CROSS JOIN UNNEST(items) AS t(i)  
WHERE i > 3

This query returns:

+------------+
<table>
<thead>
<tr>
<th>array_items</th>
</tr>
</thead>
<tbody>
<tr>
<td>[4, 5]</td>
</tr>
</tbody>
</table>
+------------+

Filter an array based on whether one of its elements contain a specific value, such as 2, as in this example:

WITH
dataset AS (  
  SELECT ARRAY  
    [  
      ARRAY [1,2,3,4],  
      ARRAY [5,6,7,8],  
      ARRAY [9,0]  
    ] AS items  
)  
SELECT i AS array_items FROM dataset  
CROSS JOIN UNNEST(items) AS t(i)  
WHERE contains(i, 2)

This query returns:

+------------+
<table>
<thead>
<tr>
<th>array_items</th>
</tr>
</thead>
<tbody>
<tr>
<td>[4, 5]</td>
</tr>
<tr>
<td>[1, 2, 3, 4, 5]</td>
</tr>
</tbody>
</table>
+------------+
Sorting arrays

To create a sorted array of unique values from a set of rows, you can use the array_sort function, as in the following example.

```
WITH dataset AS (  
    SELECT ARRAY[3,1,2,5,2,3,6,3,4,5] AS items
)  
SELECT array_sort(array_agg(distinct i)) AS array_items  
FROM dataset  
CROSS JOIN UNNEST(items) AS t(i)
```

This query returns:

```
+--------------------+
| array_items        |
+--------------------+
| [1, 2, 3, 4, 5, 6] |
+--------------------+
```

For information about expanding an array into multiple rows, see Flattening nested arrays (p. 390).

Using aggregation functions with arrays

- To add values within an array, use SUM, as in the following example.
To aggregate multiple rows within an array, use `array_agg`. For information, see Converting arrays from subqueries (p. 392).

**Note**
ORDER BY is supported for aggregation functions starting in Athena engine version 2.

```sql
WITH
dataset AS (
    SELECT ARRAY[
        ARRAY[1,2,3,4],
        ARRAY[5,6,7,8],
        ARRAY[9,0]
    ] AS items,
),
item AS (SELECT i AS array_items
    FROM dataset, UNNEST(items) AS t(i)
)
SELECT array_items, sum(val) AS total
FROM item, UNNEST(array_items) AS t(val)
GROUP BY array_items;
```

In the last SELECT statement, instead of using `sum()` and `UNNEST`, you can use `reduce()` to decrease processing time and data transfer, as in the following example.

```sql
WITH
dataset AS (
    SELECT ARRAY[
        ARRAY[1,2,3,4],
        ARRAY[5,6,7,8],
        ARRAY[9,0]
    ] AS items,
),
item AS (SELECT i AS array_items
    FROM dataset, UNNEST(items) AS t(i)
)
SELECT array_items, reduce(array_items, 0 , (s, x) -> s + x, s -> s) AS total
FROM item;
```

Either query returns the following results. The order of returned results is not guaranteed.

```
<table>
<thead>
<tr>
<th>array_items</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1, 2, 3, 4]</td>
<td>10</td>
</tr>
<tr>
<td>[5, 6, 7, 8]</td>
<td>26</td>
</tr>
<tr>
<td>[9, 0]</td>
<td>9</td>
</tr>
</tbody>
</table>
```

### Converting arrays to strings

To convert an array into a single string, use the `array_join` function. The following standalone example creates a table called `dataset` that contains an aliased array called `words`. The query uses `array_join` to join the array elements in `words`, separate them with spaces, and return the resulting string in an aliased column called `welcome_msg`.

```sql
```
WITH dataset AS {
  SELECT ARRAY ['hello', 'amazon', 'athena'] AS words
}
SELECT array_join(words, '') AS welcome_msg
FROM dataset

This query returns:

+---------------------+
| welcome_msg         |
+---------------------+
| hello amazon athena |
+---------------------+

Using arrays to create maps

Maps are key-value pairs that consist of data types available in Athena. To create maps, use the MAP operator and pass it two arrays: the first is the column (key) names, and the second is values. All values in the arrays must be of the same type. If any of the map value array elements need to be of different types, you can convert them later.

Examples

This example selects a user from a dataset. It uses the MAP operator and passes it two arrays. The first array includes values for column names, such as "first", "last", and "age". The second array consists of values for each of these columns, such as "Bob", "Smith", "35".

WITH dataset AS (
  SELECT MAP(
    ARRAY['first', 'last', 'age'],
    ARRAY['Bob', 'Smith', '35']
  ) AS user
)
SELECT user FROM dataset

This query returns:

+---------------------------------+
| user                            |
+---------------------------------+
| {last=Smith, first=Bob, age=35} |
+---------------------------------+

You can retrieve Map values by selecting the field name followed by [key_name], as in this example:

WITH dataset AS ( 
  SELECT MAP(
    ARRAY['first', 'last', 'age'],
    ARRAY['Bob', 'Smith', '35']
  ) AS user
)
SELECT user['first'] AS first_name FROM dataset

This query returns:

+------------+
| Bob        |
+------------+
Querying arrays with complex types and nested structures

Your source data often contains arrays with complex data types and nested structures. Examples in this section show how to change element's data type, locate elements within arrays, and find keywords using Athena queries.

- Creating a ROW (p. 397)
- Changing field names in arrays using CAST (p. 397)
- Filtering arrays using the . notation (p. 398)
- Filtering arrays with nested values (p. 398)
- Filtering arrays using UNNEST (p. 399)
- Finding keywords in arrays using regexp_like (p. 400)

Creating a ROW

Note
The examples in this section use ROW as a means to create sample data to work with. When you query tables within Athena, you do not need to create ROW data types, as they are already created from your data source. When you use CREATE_TABLE, Athena defines a STRUCT in it, populates it with data, and creates the ROW data type for you, for each row in the dataset. The underlying ROW data type consists of named fields of any supported SQL data types.

```
WITH dataset AS (
    SELECT
        ROW('Bob', 38) AS users
    )
SELECT * FROM dataset
```

This query returns:

```
| users                   |
+-------------------------+
| {field0=Bob, field1=38} |
+-------------------------+
```

Changing field names in arrays using CAST

To change the field name in an array that contains ROW values, you can CAST the ROW declaration:

```
WITH dataset AS ( 
    SELECT
        CAST(
            ROW('Bob', 38) AS ROW(name VARCHAR, age INTEGER)
        ) AS users 
    )
SELECT * FROM dataset
```
This query returns:

```
+------------------------+
| users                  |
+------------------------+
| {NAME=Bob, AGE=38}     |
+------------------------+
```

**Note**

In the example above, you declare name as a VARCHAR because this is its type in Presto. If you declare this STRUCT inside a CREATE TABLE statement, use String type because Hive defines this data type as String.

**Filtering arrays using the . notation**

In the following example, select the accountId field from the userIdentity column of a AWS CloudTrail logs table by using the dot . notation. For more information, see Querying AWS CloudTrail Logs (p. 454).

```
SELECT CAST(useridentity.accountid AS bigint) as newid
FROM cloudtrail_logs
LIMIT 2;
```

This query returns:

```
+------------+
| newid      |
+------------+
| 112233445566 |
| 99877665544 |
+------------+
```

To query an array of values, issue this query:

```
WITH dataset AS (
    SELECT ARRAY[
        CAST(ROW('Bob', 38) AS ROW(name VARCHAR, age INTEGER)),
        CAST(ROW('Alice', 35) AS ROW(name VARCHAR, age INTEGER)),
        CAST(ROW('Jane', 27) AS ROW(name VARCHAR, age INTEGER))
    ] AS users
)
SELECT * FROM dataset
```

It returns this result:

```
+---------------------+
| users               |
+---------------------+
| [{NAME=Bob, AGE=38}, {NAME=Alice, AGE=35}, {NAME= Jane, AGE=27}] |
+---------------------+
```

**Filtering arrays with nested values**

Large arrays often contain nested structures, and you need to be able to filter, or search, for values within them.
To define a dataset for an array of values that includes a nested BOOLEAN value, issue this query:

```sql
WITH dataset AS (  
    SELECT  
        CAST(  
            ROW('aws.amazon.com', ROW(true)) AS ROW(hostname VARCHAR, flaggedActivity ROW(isNew BOOLEAN))  
        ) AS sites  
    )  
SELECT * FROM dataset
```

It returns this result:

```
+----------------------------------------------------------+  
| sites                                                    |  
+----------------------------------------------------------+  
| {HOSTNAME=aws.amazon.com, FLAGGEDACTIVITY={ISNEW=true}}  |  
```

Next, to filter and access the BOOLEAN value of that element, continue to use the dot . notation.

```sql
WITH dataset AS (  
    SELECT  
        CAST(  
            ROW('aws.amazon.com', ROW(true)) AS ROW(hostname VARCHAR, flaggedActivity ROW(isNew BOOLEAN))  
        ) AS sites  
    )  
SELECT sites.hostname, sites.flaggedactivity.isnew  
FROM dataset
```

This query selects the nested fields and returns this result:

```
+------------------------+  
| hostname       | isnew |  
+------------------------+  
| aws.amazon.com | true  |  
```

### Filtering arrays using UNNEST

To filter an array that includes a nested structure by one of its child elements, issue a query with an UNNEST operator. For more information about UNNEST, see Flattening Nested Arrays (p. 390).

For example, this query finds host names of sites in the dataset.

```sql
WITH dataset AS (  
    SELECT ARRAY[  
        CAST(  
            ROW('aws.amazon.com', ROW(true)) AS ROW(hostname VARCHAR, flaggedActivity ROW(isNew BOOLEAN))  
        ),  
        CAST(  
            ROW('news.cnn.com', ROW(false)) AS ROW(hostname VARCHAR, flaggedActivity ROW(isNew BOOLEAN))  
        ),  
        CAST(  
            ROW('netflix.com', ROW(false)) AS ROW(hostname VARCHAR, flaggedActivity ROW(isNew BOOLEAN))  
        )  
    ] AS hosts  
    )  
SELECT hostname FROM hosts
```

```
+------------------------+  
| hostname       | isnew |  
+------------------------+  
| aws.amazon.com | true  |  
```
### Querying arrays with complex types and nested structures

```sql
SELECT sites.hostname, sites.flaggedActivity.isNew
FROM dataset, UNNEST(items) t(sites)
WHERE sites.flaggedActivity.isNew = true
```

It returns:

```
+------------------------+
| hostname       | isnew |
+------------------------+
| aws.amazon.com | true  |
+------------------------+
```

### Finding keywords in arrays using `regexp_like`

The following examples illustrate how to search a dataset for a keyword within an element inside an array, using the `regexp_like` function. It takes as an input a regular expression pattern to evaluate, or a list of terms separated by a pipe (|), evaluates the pattern, and determines if the specified string contains it.

The regular expression pattern needs to be contained within the string, and does not have to match it. To match the entire string, enclose the pattern with ^ at the beginning of it, and $ at the end, such as `'^pattern$'`.

Consider an array of sites containing their host name, and a `flaggedActivity` element. This element includes an `ARRAY`, containing several `MAP` elements, each listing different popular keywords and their popularity count. Assume you want to find a particular keyword inside a `MAP` in this array.

To search this dataset for sites with a specific keyword, we use `regexp_like` instead of the similar SQL `LIKE` operator, because searching for a large number of keywords is more efficient with `regexp_like`.

#### Example Example 1: Using `regexp_like`

The query in this example uses the `regexp_like` function to search for terms 'politics|bigdata', found in values within arrays:

```sql
WITH dataset AS (
    SELECT ARRAY[
        CAST(
            ROW('aws.amazon.com', ROW(
                ARRAY[
                    MAP(['term', 'count'], ARRAY(['bigdata', '10'])),
                    MAP(['term', 'count'], ARRAY(['serverless', '50'])),
                    MAP(['term', 'count'], ARRAY(['analytics', '82'])),
                    MAP(['term', 'count'], ARRAY(['iot', '74'])))
        ) AS ROW(hostname VARCHAR, flaggedActivity ROW(flags ARRAY(MAP(VARCHAR, VARCHAR)) ))
    ),
    CAST(
        ROW('news.cnn.com', ROW(ARRAY[
            MAP(['term', 'count'], ARRAY(['politics', '241'])),
            MAP(['term', 'count'], ARRAY(['technology', '211'])),
            MAP(['term', 'count'], ARRAY(['serverless', '25'])),
            MAP(['term', 'count'], ARRAY(['iot', '170'])))
        ) AS ROW(hostname VARCHAR, flaggedActivity ROW(flags ARRAY(MAP(VARCHAR, VARCHAR)) ))
    ),
    CAST(
        ROW('netflix.com', ROW(ARRAY[
```
Example Example 2: Using `regexp_like`

The query in the following example adds up the total popularity scores for the sites matching your search terms with the `regexp_like` function, and then orders them from highest to lowest.

```sql
WITH dataset AS (
    SELECT
        ROW('aws.amazon.com', ROW(
            MAP(['term', 'count'], ['bigdata', '10']),
            MAP(['term', 'count'], ['serverless', '50']),
            MAP(['term', 'count'], ['analytics', '82']),
            MAP(['term', 'count'], ['iot', '74'])
        ) AS ROW(hostname VARCHAR, flaggedActivity ROW(flags ARRAY(MAP(VARCHAR, VARCHAR)) ))
    ),
    CAST(
        ROW('news.cnn.com', ROW(
            MAP(['term', 'count'], ['politics', '241']),
            MAP(['term', 'count'], ['technology', '211']),
            MAP(['term', 'count'], ['serverless', '25']),
            MAP(['term', 'count'], ['iot', '170'])
        ) AS ROW(hostname VARCHAR, flaggedActivity ROW(flags ARRAY(MAP(VARCHAR, VARCHAR)) ))
    ),
    CAST(
        ROW('netflix.com', ROW(
            MAP(['term', 'count'], ['cartoons', '1020']),
            MAP(['term', 'count'], ['house of cards', '112042']),
            MAP(['term', 'count'], ['orange is the new black', '342']),
            MAP(['term', 'count'], ['iot', '4'])
        ) AS ROW(hostname VARCHAR, flaggedActivity ROW(flags ARRAY(MAP(VARCHAR, VARCHAR)) ))
    ) AS items

    SELECT hostname
    FROM sites, UNNEST(sites.flaggedActivity.flags) t(flags)
    WHERE regexp_like(flags['term'], 'politics|bigdata')
    GROUP BY hostname
)

This query returns two sites:

<table>
<thead>
<tr>
<th>hostname</th>
</tr>
</thead>
<tbody>
<tr>
<td>aws.amazon.com</td>
</tr>
<tr>
<td>news.cnn.com</td>
</tr>
</tbody>
</table>

Example Example 2: Using `regexp_like`

The query in the following example adds up the total popularity scores for the sites matching your search terms with the `regexp_like` function, and then orders them from highest to lowest.

```sql
WITH dataset AS (
    SELECT
        ROW('aws.amazon.com', ROW(
            MAP(['term', 'count'], ['bigdata', '10']),
            MAP(['term', 'count'], ['serverless', '50']),
            MAP(['term', 'count'], ['analytics', '82']),
            MAP(['term', 'count'], ['iot', '74'])
        ) AS ROW(hostname VARCHAR, flaggedActivity ROW(flags ARRAY(MAP(VARCHAR, VARCHAR)) ))
    ),
    CAST(
        ROW('news.cnn.com', ROW(
            MAP(['term', 'count'], ['politics', '241']),
            MAP(['term', 'count'], ['technology', '211']),
            MAP(['term', 'count'], ['serverless', '25']),
            MAP(['term', 'count'], ['iot', '170'])
        ) AS ROW(hostname VARCHAR, flaggedActivity ROW(flags ARRAY(MAP(VARCHAR, VARCHAR)) ))
    ),
    CAST(
        ROW('netflix.com', ROW(
            MAP(['term', 'count'], ['cartoons', '1020']),
            MAP(['term', 'count'], ['house of cards', '112042']),
            MAP(['term', 'count'], ['orange is the new black', '342']),
            MAP(['term', 'count'], ['iot', '4'])
        ) AS ROW(hostname VARCHAR, flaggedActivity ROW(flags ARRAY(MAP(VARCHAR, VARCHAR)) ))
    ) AS items

    SELECT hostname
    FROM sites, UNNEST(sites.flaggedActivity.flags) t(flags)
    WHERE regexp_like(flags['term'], 'politics|bigdata')
    GROUP BY hostname
)

This query returns two sites:

<table>
<thead>
<tr>
<th>hostname</th>
</tr>
</thead>
<tbody>
<tr>
<td>aws.amazon.com</td>
</tr>
<tr>
<td>news.cnn.com</td>
</tr>
</tbody>
</table>
```
### Querying geospatial data

Geospatial data contains identifiers that specify a geographic position for an object. Examples of this type of data include weather reports, map directions, tweets with geographic positions, store locations, and airline routes. Geospatial data plays an important role in business analytics, reporting, and forecasting.

Geospatial identifiers, such as latitude and longitude, allow you to convert any mailing address into a set of geographic coordinates.

**Topics**

- What is a geospatial query? (p. 402)
- Input data formats and geometry data types (p. 403)
- Supported geospatial functions (p. 403)
- Examples: Geospatial queries (p. 418)

#### What is a geospatial query?

Geospatial queries are specialized types of SQL queries supported in Athena. They differ from non-spatial SQL queries in the following ways:

- Using the following specialized geometry data types: point, line, multiline, polygon, and multipolygon.
- Expressing relationships between geometry data types, such as distance, equals, crosses, touches, overlaps, disjoint, and others.

Using geospatial queries in Athena, you can run these and other similar operations:

- Find the distance between two points.
- Check whether one area (polygon) contains another.
- Check whether one line crosses or touches another line or polygon.
For example, to obtain a point geometry data type from values of type double for the geographic coordinates of Mount Rainier in Athena, use the `ST_Point (longitude, latitude)` geospatial function, as in the following example.

```
ST_Point(-121.7602, 46.8527)
```

**Input data formats and geometry data types**

To use geospatial functions in Athena, input your data in the WKT format, or use the Hive JSON SerDe. You can also use the geometry data types supported in Athena.

**Input data formats**

To handle geospatial queries, Athena supports input data in these data formats:

- **WKT (Well-known Text).** In Athena, WKT is represented as a `varchar` data type.
- **JSON-encoded geospatial data.** To parse JSON files with geospatial data and create tables for them, Athena uses the Hive JSON SerDe. For more information about using this SerDe in Athena, see JSON SerDe libraries (p. 319).

**Geometry data types**

To handle geospatial queries, Athena supports these specialized geometry data types:

- point
- line
- polygon
- multiline
- multipolygon

**Supported geospatial functions**

The geospatial functions that are available in Athena depend on the engine version that you use.

- For information about the geospatial functions in Athena engine version 3, see Geospatial functions in the Trino documentation.
- For a list of function name changes and new functions as of Athena engine version 2, see Geospatial function name changes and new functions in Athena engine version 2 (p. 416).

For information about Athena engine versioning, see Athena engine versioning (p. 669).

**Topics**

- Geospatial functions in Athena engine version 3 (p. 403)
- Geospatial functions in Athena engine version 2 (p. 404)

**Geospatial functions in Athena engine version 3**

For information about the geospatial functions in Athena engine version 3, see Geospatial functions in the Trino documentation.
Geospatial functions in Athena engine version 2

This topic lists the ESRI geospatial functions that are supported starting in Athena engine version 2. For information about Athena engine versions, see Athena engine versioning (p. 669).

Changes in Athena engine version 2

- The input and output types for some functions have changed. Most notably, the VARBINARY type is no longer directly supported for input. For more information, see Changes to geospatial functions (p. 687).
- The names of some geospatial functions have changed. For more information, see Geospatial function name changes in Athena engine version 2 (p. 416).
- New functions have been added. For more information, see New geospatial functions in Athena engine version 2 (p. 417).

Athena supports the following types of geospatial functions:

- Constructor functions (p. 404)
- Geospatial relationship functions (p. 406)
- Operation functions (p. 408)
- Accessor functions (p. 410)
- Aggregation functions (p. 415)
- Bing tile functions (p. 415)

Constructor functions

Use constructor functions to obtain binary representations of point, line, or polygon geometry data types. You can also use these functions to convert binary data to text, and obtain binary values for geometry data that is expressed as Well-Known Text (WKT).

**ST_AsBinary(geometry)**

Returns a varbinary data type that contains the WKB representation of the specified geometry. Example:

```
SELECT ST_AsBinary(ST_Point(-158.54, 61.56))
```

**ST_AsText(geometry)**

Converts each of the specified geometry data types (p. 403) to text. Returns a value in a varchar data type, which is a WKT representation of the geometry data type. Example:

```
SELECT ST_AsText(ST_Point(-158.54, 61.56))
```

**ST_GeomAsLegacyBinary(geometry)**

Returns a legacy varbinary from the specified geometry. Example:

```
SELECT ST_GeomAsLegacyBinary(ST_Point(-158.54, 61.56))
```

**ST_GeometryFromText(varchar)**

Converts text in WKT format into a geometry data type. Returns a value in a geometry data type. Example:
Supported geospatial functions

**SELECT ST_GeometryFromText(ST_AsText(ST_Point(1, 2)))**

**ST_GeomFromBinary(varbinary)**

Returns a geometry type object from a WKB representation. Example:

**SELECT ST_GeomFromBinary(ST_AsBinary(ST_Point(-158.54, 61.56)))**

**ST_GeomFromLegacyBinary(varbinary)**

Returns a geometry type object from a legacy varbinary type. Example:

**SELECT ST_GeomFromLegacyBinary(ST_GeomAsLegacyBinary(ST_Point(-158.54, 61.56)))**

**ST_LineFromText(varchar)**

Returns a value in the geometry data type (p. 403) line. Example:

**SELECT ST_Line('linestring(1 1, 2 2, 3 3)')**

**ST_LineString(array(point))**

Returns a LineString geometry type formed from an array of point geometry types. If there are fewer than two non-empty points in the specified array, an empty LineString is returned. Throws an exception if any element in the array is null, empty, or the same as the previous one. The returned geometry may not be simple. Depending on the input specified, the returned geometry can self-intersect or contain duplicate vertexes. Example:

**SELECT ST_LineString(ARRAY[ST_Point(-158.54, 61.56), ST_Point(-158.55, 61.56)])**

**ST_MultiPoint(array(point))**

Returns a MultiPoint geometry object formed from the specified points. Returns null if the specified array is empty. Throws an exception if any element in the array is null or empty. The returned geometry may not be simple and can contain duplicate points if the specified array has duplicates. Example:

**SELECT ST_MultiPoint(ARRAY[ST_Point(-158.54, 61.56), ST_Point(-158.55, 61.56)])**

**ST_Point(double, double)**

Returns a geometry type point object. For the input data values to this function, use geometric values, such as values in the Universal Transverse Mercator (UTM) Cartesian coordinate system, or geographic map units (longitude and latitude) in decimal degrees. The longitude and latitude values use the World Geodetic System, also known as WGS 1984, or EPSG:4326. WGS 1984 is the coordinate system used by the Global Positioning System (GPS).

For example, in the following notation, the map coordinates are specified in longitude and latitude, and the value .072284, which is the buffer distance, is specified in angular units as decimal degrees:

**SELECT ST_Buffer(ST_Point(-74.006801, 40.705220), .072284)**

Syntax:
SELECT ST_Point(longitude, latitude) FROM earthquakes LIMIT 1

The following example uses specific longitude and latitude coordinates:

```
SELECT ST_Point(-158.54, 61.56)
FROM earthquakes
LIMIT 1
```

The next example uses specific longitude and latitude coordinates:

```
SELECT ST_Point(-74.006801, 40.705220)
```

The following example uses the `ST_AsText` function to obtain the geometry from WKT:

```
SELECT ST_AsText(ST_Point(-74.006801, 40.705220)) AS WKT
```

**`ST_Polygon(varchar)`**

Using the sequence of the ordinates provided clockwise, left to right, returns a geometry data type (p. 403) polygon. Starting in Athena engine version 2, only polygons are accepted as inputs. Example:

```
SELECT ST_Polygon('polygon ((1 1, 1 4, 4 4, 4 1))')
```

**`to_geometry(sphericalGeography)`**

Returns a geometry object from the specified spherical geography object. Example:

```
SELECT to_geometry(to_spherical_geography(ST_Point(-158.54, 61.56)))
```

**`to_spherical_geography(geometry)`**

Returns a spherical geography object from the specified geometry. Use this function to convert a geometry object to a spherical geography object on the sphere of the Earth's radius. This function can be used only on POINT, MULTIPoint, LINESTRING, MULTILINESTRING, POLYGON, and MULTIPOLYGON geometries defined in 2D space or a GEOMETRYCOLLECTION of such geometries. For each point of the specified geometry, the function verifies that point.x is within [-180.0, 180.0] and point.y is within [-90.0, 90.0]. The function uses these points as longitude and latitude degrees to construct the shape of the sphericalGeography result. Example:

```
SELECT to_spherical_geography(ST_Point(-158.54, 61.56))
```

**Geospatial relationship functions**

The following functions express relationships between two different geometries that you specify as input and return results of type boolean. The order in which you specify the pair of geometries matters: the first geometry value is called the left geometry, the second geometry value is called the right geometry. These functions return:

- TRUE if and only if the relationship described by the function is satisfied.
- FALSE if and only if the relationship described by the function is not satisfied.
ST_Contains(geometry, geometry)

Returns TRUE if and only if the left geometry contains the right geometry. Examples:

<table>
<thead>
<tr>
<th>SQL Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT ST_Contains('POLYGON((0 2,1 1,0 -1,0 2))', 'POLYGON((-1 3,2 1,0 -3,-1 3))')</td>
</tr>
<tr>
<td>SELECT ST_Contains('POLYGON((0 2,1 1,0 -1,0 2))', ST_Point(0, 0))</td>
</tr>
<tr>
<td>SELECT ST_Contains(ST_GeometryFromText('POLYGON((0 2,1 1,0 -1,0 2))'),</td>
</tr>
<tr>
<td>ST_GeometryFromText('POLYGON((-1 3,2 1,0 -3,-1 3))'))</td>
</tr>
</tbody>
</table>

ST_Crosses(geometry, geometry)

Returns TRUE if and only if the left geometry crosses the right geometry. Example:

<table>
<thead>
<tr>
<th>SQL Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT ST_Crosses(ST_Line('linestring(1 1, 2 2)'), ST_Line('linestring(0 1, 2 2)'))</td>
</tr>
</tbody>
</table>

ST_Disjoint(geometry, geometry)

Returns TRUE if and only if the intersection of the left geometry and the right geometry is empty. Example:

<table>
<thead>
<tr>
<th>SQL Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT ST_Disjoint(ST_Line('linestring(0 0, 0 1)'), ST_Line('linestring(1 1, 1 0)'))</td>
</tr>
</tbody>
</table>

ST_Equals(geometry, geometry)

Returns TRUE if and only if the left geometry equals the right geometry. Example:

<table>
<thead>
<tr>
<th>SQL Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT ST_Equals(ST_Line('linestring(0 0, 1 1)'), ST_Line('linestring(1 3, 2 2)'))</td>
</tr>
</tbody>
</table>

ST_Intersects(geometry, geometry)

Returns TRUE if and only if the left geometry intersects the right geometry. Example:

<table>
<thead>
<tr>
<th>SQL Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT ST_Intersects(ST_Line('linestring(8 7, 7 8)'), ST_Polygon('polygon((1 1, 4 1, 4 4, 1 4))'))</td>
</tr>
</tbody>
</table>

ST_Overlaps(geometry, geometry)

Returns TRUE if and only if the left geometry overlaps the right geometry. Example:

<table>
<thead>
<tr>
<th>SQL Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT ST_Overlaps(ST_Polygon('polygon((2 0, 2 1, 3 1))'),</td>
</tr>
<tr>
<td>ST_Polygon('polygon((1 1, 1 4, 4 4, 4 1))'))</td>
</tr>
</tbody>
</table>

ST_Relate(geometry, geometry, varchar)

Returns TRUE if and only if the left geometry has the specified dimensionally extended nine-intersection model (DE-9IM) relationship with the right geometry. The third (varchar) input takes the relationship. Example:

<table>
<thead>
<tr>
<th>SQL Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT ST_Relate(ST_Line('linestring(0 0, 3 3)'),</td>
</tr>
<tr>
<td>ST_Line('linestring(1 1, 4 4)'),</td>
</tr>
<tr>
<td>'T**********')</td>
</tr>
</tbody>
</table>
**ST_Touches(geometry, geometry)**

Returns TRUE if and only if the left geometry touches the right geometry.

Example:

```sql
SELECT ST_Touches(ST_Point(8, 8), ST_Polygon('polygon((1 1 4 4 4 1))'))
```

**ST_Within(geometry, geometry)**

Returns TRUE if and only if the left geometry is within the right geometry.

Example:

```sql
SELECT ST_Within(ST_Point(8, 8), ST_Polygon('polygon((1 1 4 4 4 1))'))
```

**Operation functions**

Use operation functions to perform operations on geometry data type values. For example, you can obtain the boundaries of a single geometry data type; intersections between two geometry data types; difference between left and right geometries, where each is of the same geometry data type; or an exterior buffer or ring around a particular geometry data type.

**geometry_union(array(geometry))**

Returns a geometry that represents the point set union of the specified geometries. Example:

```sql
SELECT geometry_union(ARRAY[ST_Point(-158.54, 61.56), ST_Point(-158.55, 61.56)])
```

**ST_Boundary(geometry)**

Takes as an input one of the geometry data types and returns the boundary geometry data type.

Examples:

```sql
SELECT ST_Boundary(ST_Line('linestring(0 1, 1 0)'))
```

```sql
SELECT ST_Boundary(ST_Polygon('polygon((1 1 4 4 4 1))'))
```

**ST_Buffer(geometry, double)**

Takes as an input one of the geometry data types, such as point, line, polygon, multiline, or multipolygon, and a distance as type `double`). Returns the geometry data type buffered by the specified distance (or radius). Example:

```sql
SELECT ST_Buffer(ST_Point(1, 2), 2.0)
```

In the following example, the map coordinates are specified in longitude and latitude, and the value \(0.072284\), which is the buffer distance, is specified in angular units as decimal degrees:

```sql
SELECT ST_Buffer(ST_Point(-74.006801, 40.705220), 0.072284)
```

**ST_Difference(geometry, geometry)**

Returns a geometry of the difference between the left geometry and right geometry. Example:
SELECT ST_AsText(ST_Difference(ST_Polygon('polygon((0 0, 0 10, 10 10, 10 0))'),
ST_Polygon('polygon((0 0, 0 5, 5 5, 5 0))')))

**ST_Envelope( geometry )**

Takes as an input line, polygon, multiline, and multipolygon geometry data types. Does not support point geometry data type. Returns the envelope as a geometry, where an envelope is a rectangle around the specified geometry data type. Examples:

```sql
SELECT ST_Envelope(ST_Linestring('linestring(0 1, 1 0)'))
```

```sql
SELECT ST_Envelope(ST_Polygon('polygon((1 1, 1 4, 4 4, 4 1))'))
```

**ST_EnvelopeAsPts( geometry )**

Returns an array of two points that represent the lower left and upper right corners of a geometry's bounding rectangular polygon. Returns null if the specified geometry is empty. Example:

```sql
SELECT ST_EnvelopeAsPts(ST_Point(-158.54, 61.56))
```

**ST_ExteriorRing( geometry )**

Returns the geometry of the exterior ring of the input type polygon. Starting in Athena engine version 2, polygons are the only geometries accepted as inputs. Examples:

```sql
SELECT ST_ExteriorRing(ST_Polygon(1,1, 1,4, 4,1))
```

```sql
SELECT ST_ExteriorRing(ST_Polygon('polygon((0 0, 8 0, 0 8, 0 0), (1 1, 1 5, 5 1, 1 1))'))
```

**ST_Intersection( geometry, geometry )**

Returns the geometry of the intersection of the left geometry and right geometry. Examples:

```sql
SELECT ST_Intersection(ST_Point(1,1), ST_Point(1,1))
```

```sql
SELECT ST_Intersection(ST_Linestring('linestring(0 1, 1 0)'), ST_Polygon('polygon((1 1, 1 4, 4 4, 4 1))'))
```

```sql
SELECT ST_AsText(ST_Intersection(ST_Polygon('polygon((2 0, 2 3, 3 0))'),
ST_Polygon('polygon((1 1, 4 1, 4 4, 1 4))')))
```

**ST_SymDifference( geometry, geometry )**

Returns the geometry of the geometrically symmetric difference between the left geometry and the right geometry. Example:

```sql
SELECT ST_AsText(ST_SymDifference(ST_Linestring('linestring(0 2, 2 2)'), ST_Linestring('linestring(1 2, 3 2)')))
```

**ST_Union( geometry, geometry )**

Returns a geometry data type that represents the point set union of the specified geometries. Example:
SELECT ST_Union(ST_Point(-158.54, 61.56),ST_LineString(array[ST_Point(1,2), ST_Point(3,4)]))

**Accessor functions**

Accessor functions are useful to obtain values in types varchar, bigint, or double from different geometry data types, where geometry is any of the geometry data types supported in Athena: point, line, polygon, multilinestring, and multipolygon. For example, you can obtain an area of a polygon geometry data type, maximum and minimum X and Y values for a specified geometry data type, obtain the length of a line, or receive the number of points in a specified geometry data type.

**geometry_invalid_reason(geometric)**

Returns, in a varchar data type, the reason why the specified geometry is not valid or not simple. If the specified geometry is neither valid nor simple, returns the reason why it is not valid. If the specified geometry is valid and simple, returns null. Example:

SELECT geometry_invalid_reason(ST_Point(-158.54, 61.56))

**great_circle_distance(latitude1, longitude1, latitude2, longitude2)**

Returns, as a double, the great-circle distance between two points on Earth's surface in kilometers. Example:

SELECT great_circle_distance(36.12, -86.67, 33.94, -118.40)

**line_locate_point(lineString, point)**

Returns a double between 0 and 1 that represents the location of the closest point on the specified line string to the specified point as a fraction of total 2d line length.

Returns null if the specified line string or point is empty or null. Example:

SELECT line_locate_point(ST_GeometryFromText('LINESTRING (0 0, 0 1)'), ST_Point(0, 0.2))

**simplifyGeometry(geometry, double)**

Uses the Ramer-douglas-peucker algorithm to return a geometry data type that is a simplified version of the specified geometry. Avoids creating derived geometries (in particular, polygons) that are invalid. Example:

SELECT simplify_geometry(ST_GeometryFromText('POLYGON ((1 0, 2 1, 3 1, 3 1, 4 1, 1 0))'), 1.5)

**ST_Area(geometry)**

Takes as an input a geometry data type and returns an area in type double. Example:

SELECT ST_Area(ST_Polygon('polygon((1 1, 4 1, 4 4, 1 4))'))

**ST_Centroid(geometry)**

Takes as an input a geometry data type (p. 403) polygon, and returns a point geometry data type that is the center of the polygon's envelope. Examples:
SELECT ST_Centroid(ST_GeometryFromText('polygon ((0 0, 3 6, 6 0, 0 0))'))

SELECT ST_AsText(ST_Centroid(ST_Envelope(ST_GeometryFromText('POINT (53 27)'))))

**ST_ConvexHull(geometry)**

Returns a geometry data type that is the smallest convex geometry that encloses all geometries in the specified input. Example:

SELECT ST_ConvexHull(ST_Point(-158.54, 61.56))

**ST_CoordDim(geometry)**

Takes as input one of the supported geometry data types (p. 403), and returns the count of coordinate components in the type tinyint. Example:

SELECT ST_CoordDim(ST_Point(1.5,2.5))

**ST_Dimension(geometry)**

Takes as an input one of the supported geometry data types (p. 403), and returns the spatial dimension of a geometry in type tinyint. Example:

SELECT ST_Dimension(ST_Polygon('polygon((1 1, 4 1, 4 4, 1 4))'))

**ST_Distance(geometry, geometry)**

Returns, based on spatial ref, a double containing the two-dimensional minimum Cartesian distance between two geometries in projected units. Starting in Athena engine version 2, returns null if one of the inputs is an empty geometry. Example:

SELECT ST_Distance(ST_Point(0.0,0.0), ST_Point(3.0,4.0))

**ST_Distance(sphericalGeography, sphericalGeography)**

Returns, as a double, the great-circle distance between two spherical geography points in meters. Example:

SELECT ST_Distance(to_spherical_geography(ST_Point(61.56, -86.67)), to_spherical_geography(ST_Point(61.56, -86.68)))

**ST_EndPoint(geometry)**

Returns the last point of a line geometry data type in a point geometry data type. Example:

SELECT ST_EndPoint(ST_Line('linestring(0 2, 2 2)'))

**ST_Geometries(geometry)**

Returns an array of geometries in the specified collection. If the specified geometry is not a multi-geometry, returns a one-element array. If the specified geometry is empty, returns null.

For example, given a MultiLineString object, ST_Geometries creates an array of LineString objects. Given a GeometryCollection object, ST_Geometries returns an un-flattened array of its constituents. Example:
SELECT ST_Geometries(GEOMETRYCOLLECTION(MULTIPOINT(0 0, 1 1),
GEOMETRYCOLLECTION(MULTILINESTRING((2 2, 3 3)))))

Result:

array[MULTIPOINT(0 0, 1 1),GEOMETRYCOLLECTION(MULTILINESTRING((2 2, 3 3)))]

**ST_GeometryN(geometry, index)**

Returns, as a geometry data type, the geometry element at a specified integer index. Indices start at 1. If the specified geometry is a collection of geometries (for example, a GEOMETRYCOLLECTION or MULTI* object), returns the geometry at the specified index. If the specified index is less than 1 or greater than the total number of elements in the collection, returns null. To find the total number of elements, use `ST_NumGeometries` (p. 413). Singular geometries (for example, POINT, LINESTRING, or POLYGON), are treated as collections of one element. Empty geometries are treated as empty collections. Example:

```
SELECT ST_GeometryN(ST_Point(-158.54, 61.56),1)
```

**ST_GeometryType(geometry)**

Returns, as a varchar, the type of the geometry. Example:

```
SELECT ST_GeometryType(ST_Point(-158.54, 61.56))
```

**ST_InteriorRingN(geometry, index)**

Returns the interior ring element at the specified index (indices start at 1). If the given index is less than 1 or greater than the total number of interior rings in the specified geometry, returns null. Throws an error if the specified geometry is not a polygon. To find the total number of elements, use `ST_NumInteriorRing` (p. 413). Example:

```
SELECT ST_InteriorRingN(st_polygon('polygon ((0 0, 1 0, 1 1, 0 1, 0 0))'),1)
```

**ST_InteriorRings(geometry)**

Returns a geometry array of all interior rings found in the specified geometry, or an empty array if the polygon has no interior rings. If the specified geometry is empty, returns null. If the specified geometry is not a polygon, throws an error. Example:

```
SELECT ST_InteriorRings(st_polygon('polygon ((0 0, 1 0, 1 1, 0 1, 0 0))'))
```

**ST_IsClosed(geometry)**

Takes as an input only line and multiline geometry data types (p. 403). Returns TRUE (type boolean) if and only if the line is closed. Example:

```
SELECT ST_IsClosed(ST_Line('linestring(0 2, 2 2)'))
```

**ST_IsEmpty(geometry)**

Takes as an input only line and multiline geometry data types (p. 403). Returns TRUE (type boolean) if and only if the specified geometry is empty, in other words, when the line start and end values coincide. Example:
**SELECT ST_IsEmpty(ST_Point(1.5, 2.5))**

**ST_IsRing(\text{geometry})**

Returns TRUE (type boolean) if and only if the line type is closed and simple. Example:

**SELECT ST_IsRing(ST_Line('linestring(0 2, 2 2)'))**

**ST_IsSimple(\text{geometry})**

Returns true if the specified geometry has no anomalous geometric points (for example, self intersection or self tangency). To determine why the geometry is not simple, use \texttt{geometry_invalid_reason()} (p. 410). Example:

**SELECT ST_IsSimple(ST_LineString(array[ST_Point(1,2), ST_Point(3,4)])**

**ST_IsValid(\text{geometry})**

Returns true if and only if the specified geometry is well formed. To determine why the geometry is not well formed, use \texttt{geometry_invalid_reason()} (p. 410). Example:

**SELECT ST_IsValid(ST_Point(61.56, -86.68))**

**ST_Length(\text{geometry})**

Returns the length of line in type double. Example:

**SELECT ST_Length(ST_Line('linestring(0 2, 2 2)'))**

**ST_NumGeometries(\text{geometry})**

Returns, as an integer, the number of geometries in the collection. If the geometry is a collection of geometries (for example, a GEOMETRYCOLLECTION or MULTI* object), returns the number of geometries. Single geometries return 1; empty geometries return 0. An empty geometry in a GEOMETRYCOLLECTION object counts as one geometry. For example, the following example evaluates to 1:

**ST_NumGeometries(ST_GeometryFromText('GEOMETRYCOLLECTION(MULTIPOINT EMPTY)'))**

**ST_NumInteriorRing(\text{geometry})**

Returns the number of interior rings in the polygon geometry in type bigint. Example:

**SELECT ST_NumInteriorRing(ST_Polygon('polygon ((0 0, 8 0, 0 8, 0 0), (1 1, 1 5, 5 1, 1 1)))**

**ST_NumPoints(\text{geometry})**

Returns the number of points in the geometry in type bigint. Example:

**SELECT ST_NumPoints(ST_Point(1.5, 2.5))**
**ST_PointN(lineString, index)**

Returns, as a point geometry data type, the vertex of the specified line string at the specified integer index. Indices start at 1. If the given index is less than 1 or greater than the total number of elements in the collection, returns null. To find the total number of elements, use `ST_NumPoints (p. 413)`. Example:

```
SELECT ST_PointN(ST_LineString(array[ST_Point(1,2), ST_Point(3,4)]),1)
```

**ST_Points(geometry)**

Returns an array of points from the specified line string geometry object. Example:

```
SELECT ST_Points(ST_LineString(array[ST_Point(1,2), ST_Point(3,4)]))
```

**ST_StartPoint(geometry)**

Returns the first point of a line geometry data type in a point geometry data type. Example:

```
SELECT ST_StartPoint(ST_Line('linestring(0 2, 2 2)'))
```

**ST_X(point)**

Returns the X coordinate of a point in type double. Example:

```
SELECT ST_X(ST_Point(1.5, 2.5))
```

**ST_XMax(geometry)**

Returns the maximum X coordinate of a geometry in type double. Example:

```
SELECT ST_XMax(ST_Line('linestring(0 2, 2 2)'))
```

**ST_XMin(geometry)**

Returns the minimum X coordinate of a geometry in type double. Example:

```
SELECT ST_XMin(ST_Line('linestring(0 2, 2 2)'))
```

**ST_Y(point)**

Returns the Y coordinate of a point in type double. Example:

```
SELECT ST_Y(ST_Point(1.5, 2.5))
```

**ST_YMax(geometry)**

Returns the maximum Y coordinate of a geometry in type double. Example:

```
SELECT ST_YMax(ST_Line('linestring(0 2, 2 2)'))
```

**ST_YMin(geometry)**

Returns the minimum Y coordinate of a geometry in type double. Example:
SELECT ST_YMin(ST_Line('linestring(0 2, 2 2)'))

**Aggregation functions**

*convex_hull_agg(geometry)*

Returns the minimum convex geometry that encloses all geometries passed as input.

*geometry_union_agg(geometry)*

Returns a geometry that represents the point set union of all geometries passed as input.

**Bing tile functions**

The following functions convert between geometries and tiles in the Microsoft Bing maps tile system.

*bing_tile(x, y, zoom_level)*

Returns a Bing tile object from integer coordinates $x$ and $y$ and the specified zoom level. The zoom level must be an integer from 1 through 23. Example:

```sql
SELECT bing_tile(10, 20, 12)
```

*bing_tile(quadKey)*

Returns a Bing tile object from a quadkey. Example:

```sql
SELECT bing_tile(bing_tile_quadkey(bing_tile(10, 20, 12)))
```

*bing_tile_at(latitude, longitude, zoom_level)*

Returns a Bing tile object at the specified latitude, longitude, and zoom level. The latitude must be between -85.05112878 and 85.05112878. The longitude must be between -180 and 180. The latitude and longitude values must be double and zoom_level an integer. Example:

```sql
SELECT bing_tile_at(37.431944, -122.166111, 12)
```

*bing_tiles_around(latitude, longitude, zoom_level)*

Returns an array of Bing tiles that surround the specified latitude and longitude point at the specified zoom level. Example:

```sql
SELECT bing_tiles_around(47.265511, -122.465691, 14)
```

*bing_tiles_around(latitude, longitude, zoom_level, radius_in_km)*

Returns, at the specified zoom level, an array of Bing tiles. The array contains the minimum set of Bing tiles that covers a circle of the specified radius in kilometers around the specified latitude and longitude. The latitude, longitude, and radius_in_km values are double; the zoom level is an integer. Example:

```sql
SELECT bing_tiles_around(37.8475, 112.596667, 10, .5)
```

*bing_tile_coordinates(tile)*

Returns the $x$ and $y$ coordinates of the specified Bing tile. Example:
SELECT bing_tile_coordinates(bing_tile_at(37.431944, -122.166111, 12))

**bing_tile_polygon(tile)**
Returns the polygon representation of the specified Bing tile. Example:

SELECT bing_tile_polygon(bing_tile_at(47.265511, -122.465691, 4))

**bing_tile_quadkey(tile)**
Returns the quadkey of the specified Bing tile. Example:

SELECT bing_tile_quadkey(bing_tile(52, 143, 10))

**bing_tile_zoom_level(tile)**
Returns the zoom level of the specified Bing tile as an integer. Example:

SELECT bing_tile_zoom_level(bing_tile(52, 143, 10))

**geometry_to_bing_tiles(geometry, zoom_level)**
Returns the minimum set of Bing tiles that fully covers the specified geometry at the specified zoom level. Zoom levels from 1 to 23 are supported. Example:

SELECT geometry_to_bing_tiles(ST_Point(61.56, 58.54), 10)

**Geospatial function name changes and new functions in Athena engine version 2**

This section lists changes in geospatial function names and geospatial functions that are new in Athena engine version 2. For information about other changes as of Athena engine version 2, see Athena engine version 2 (p. 680).

For information about Athena engine versioning, see Athena engine versioning (p. 669).

**Geospatial function name changes in Athena engine version 2**

The names of the following functions have changed. In some cases, the input and output types have also changed. For more information, visit the corresponding links.

<table>
<thead>
<tr>
<th>Prior function name</th>
<th>Function name starting in Athena engine version 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>st_coordinate_dimension</td>
<td>ST_CoordDim (p. 411)</td>
</tr>
<tr>
<td>st_end_point</td>
<td>ST_EndPoint (p. 411)</td>
</tr>
<tr>
<td>st_exterior_ring</td>
<td>ST_ExteriorRing (p. 409)</td>
</tr>
<tr>
<td>st_interior_ring_number</td>
<td>ST_NumInteriorRing (p. 413)</td>
</tr>
<tr>
<td>st_geometry_from_text</td>
<td>ST_GeometryFromText (p. 404)</td>
</tr>
<tr>
<td>st_is_closed</td>
<td>ST_IsClosed (p. 412)</td>
</tr>
<tr>
<td>Prior function name</td>
<td>Function name starting in Athena engine version 2</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>st_is_empty</td>
<td>ST_IsEmpty (p. 412)</td>
</tr>
<tr>
<td>st_is_ring</td>
<td>ST_IsRing (p. 413)</td>
</tr>
<tr>
<td>st_max_x</td>
<td>ST_XMax (p. 414)</td>
</tr>
<tr>
<td>st_max_y</td>
<td>ST_YMax (p. 414)</td>
</tr>
<tr>
<td>st_min_x</td>
<td>ST_XMin (p. 414)</td>
</tr>
<tr>
<td>st_min_y</td>
<td>ST_YMin (p. 414)</td>
</tr>
<tr>
<td>st_point_number</td>
<td>ST_NumPoints (p. 413)</td>
</tr>
<tr>
<td>st_start_point</td>
<td>ST_StartPoint (p. 414)</td>
</tr>
<tr>
<td>st_symmetric_difference</td>
<td>ST_SymDifference (p. 409)</td>
</tr>
</tbody>
</table>

**New geospatial functions in Athena engine version 2**

The following geospatial functions are new as of Athena engine version 2. For more information, visit the corresponding links.

**Constructor functions**

- ST_AsBinary (p. 404)
- ST_GeomAsLegacyBinary (p. 404)
- ST_GeomFromBinary (p. 405)
- ST_GeomFromLegacyBinary (p. 405)
- ST_LineString (p. 405)
- ST_MultiPoint (p. 405)
- to_geometry (p. 406)
- to_spherical_geography (p. 406)

**Operation functions**

- geometry_union (p. 408)
- ST_EnvelopeAsPts (p. 409)
- ST_Union (p. 409)

**Accessor functions**

- geometry_invalid_reason (p. 410)
- great_circle_distance (p. 410)
- line_locate_point (p. 410)
- simplify_geometry (p. 410)
- ST_ConvexHull (p. 411)
- ST_Distance (spherical geography) (p. 411)
- ST_Geometries (p. 411)
- ST_GeometryN (p. 412)
- ST_GeometryType (p. 412)
Examples: Geospatial queries

The examples in this topic create two tables from sample data available on GitHub and query the tables based on the data. The sample data, which are for illustration purposes only and are not guaranteed to be accurate, are in the following files:

- **earthquakes.csv** – Lists earthquakes that occurred in California. The example earthquakes table uses fields from this data.
- **california-counties.json** – Lists county data for the state of California in ESRI-compliant GeoJSON format. The data includes many fields such as AREA, PERIMETER, STATE, COUNTY, and NAME, but the example counties table uses only two: Name (string), and BoundaryShape (binary).

  **Note**
  Athena uses the com.esri.json.hadoop.EnclosedJsonInputFormat to convert the JSON data to geospatial binary format.

The following code example creates a table called `earthquakes`:

```sql
CREATE external TABLE earthquakes
(
  earthquake_date string,
  latitude double,
  longitude double,
  depth double,
  magnitude double,
```
magtype string,
mbstations string,
gap string,
distance string,
rms string,
source string,
eventid string
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
STORED AS TEXTFILE LOCATION 's3://my-query-log/csv/';

The following code example creates a table called counties:

CREATE external TABLE IF NOT EXISTS counties
(
  Name string,
  BoundaryShape binary
)
ROW FORMAT SERDE 'com.esri.hadoop.hive.serde.JsonSerde'
STORED AS INPUTFORMAT 'com.esri.json.hadoop.EnclosedJsonInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION 's3://my-query-log/json/';

The following example query uses the CROSS JOIN function on the counties and earthquake tables. The example uses ST_CONTAINS to query for counties whose boundaries include earthquake locations, which are specified with ST_POINT. The query groups such counties by name, orders them by count, and returns them in descending order.

**Note**

Starting in Athena engine version 2, functions like ST_CONTAINS no longer support the VARBINARY type as an input. For this reason, the example uses the ST_GeomFromLegacyBinary(varbinary) (p. 405) function to convert the boundaryshape binary value into a geometry. For more information, see Changes to geospatial functions (p. 687) in the Athena engine version 2 (p. 680) reference.

```sql
SELECT counties.name,
   COUNT(*) cnt
FROM counties
CROSS JOIN earthquakes
WHERE ST_CONTAINS (ST_GeomFromLegacyBinary(counties.boundaryshape),
  ST_POINT(earthquakes.longitude, earthquakes.latitude))
GROUP BY  counties.name
ORDER BY  cnt DESC
```

This query returns:

```
+------------------------+----+
| name             | cnt |
+------------------------+----+
| Kern             | 36  |
+------------------------+----+
| San Bernardino   | 35  |
+------------------------+----+
| Imperial         | 28  |
+------------------------+----+
| Inyo             | 20  |
+------------------------+----+
| Los Angeles      | 18  |
+------------------------+----+
| Riverside        | 14  |
+------------------------+----+
```
<table>
<thead>
<tr>
<th>Monterey</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Santa Clara</td>
<td>12</td>
</tr>
<tr>
<td>San Benito</td>
<td>11</td>
</tr>
<tr>
<td>Fresno</td>
<td>11</td>
</tr>
<tr>
<td>San Diego</td>
<td>7</td>
</tr>
<tr>
<td>Santa Cruz</td>
<td>5</td>
</tr>
<tr>
<td>Ventura</td>
<td>3</td>
</tr>
<tr>
<td>San Luis Obispo</td>
<td>3</td>
</tr>
<tr>
<td>Orange</td>
<td>2</td>
</tr>
<tr>
<td>San Mateo</td>
<td>1</td>
</tr>
</tbody>
</table>

**Additional resources**

For additional examples of geospatial queries, see the following blog posts:

- Querying OpenStreetMap with Amazon Athena
- Visualize over 200 years of global climate data using Amazon Athena and Amazon QuickSight.

**Using Athena to query Apache Hudi datasets**

*Apache Hudi* is an open-source data management framework that simplifies incremental data processing. Record-level insert, update, upsert, and delete actions are processed much more granularly, reducing overhead. **Upsert** refers to the ability to insert records into an existing dataset if they do not already exist or to update them if they do.

Hudi handles data insertion and update events without creating many small files that can cause performance issues for analytics. Apache Hudi automatically tracks changes and merges files so that they remain optimally sized. This avoids the need to build custom solutions that monitor and re-write many small files into fewer large files.

Hudi datasets are suitable for the following use cases:

- Complying with privacy regulations like General data protection regulation (GDPR) and California consumer privacy act (CCPA) that enforce people's right to remove personal information or change how their data is used.
- Working with streaming data from sensors and other Internet of Things (IoT) devices that require specific data insertion and update events.
- Implementing a change data capture (CDC) system.

Data sets managed by Hudi are stored in S3 using open storage formats. Currently, Athena can read compacted Hudi datasets but not write Hudi data. Athena uses Apache Hudi version 0.8.0, subject to change. For more information about this Hudi version, see [Release 0.8.0 (docs)] on the Apache website.

**Hudi dataset table types**

A Hudi dataset can be one of the following types:
Hudi dataset table types

- **Copy on Write (CoW)** – Data is stored in a columnar format (Parquet), and each update creates a new version of files during a write.
- **Merge on Read (MoR)** – Data is stored using a combination of columnar (Parquet) and row-based (Avro) formats. Updates are logged to row-based delta files and are compacted as needed to create new versions of the columnar files.

With CoW datasets, each time there is an update to a record, the file that contains the record is rewritten with the updated values. With a MoR dataset, each time there is an update, Hudi writes only the row for the changed record. MoR is better suited for write- or change-heavy workloads with fewer reads. CoW is better suited for read-heavy workloads on data that change less frequently.

Hudi provides three query types for accessing the data:

- **Snapshot queries** – Queries that see the latest snapshot of the table as of a given commit or compaction action. For MoR tables, snapshot queries expose the most recent state of the table by merging the base and delta files of the latest file slice at the time of the query.
- **Incremental queries** – Queries only see new data written to the table, since a given commit/compaction. This effectively provides change streams to enable incremental data pipelines.
- **Read optimized queries** – For MoR tables, queries see the latest data compacted. For CoW tables, queries see the latest data committed.

The following table shows the possible Hudi query types for each table type.

<table>
<thead>
<tr>
<th>Table type</th>
<th>Possible Hudi query types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy On Write</td>
<td>snapshot, incremental</td>
</tr>
<tr>
<td>Merge On Read</td>
<td>snapshot, incremental, read optimized</td>
</tr>
</tbody>
</table>

Currently, Athena supports snapshot queries and read optimized queries, but not incremental queries. On MoR tables, all data exposed to read optimized queries are compacted. This provides good performance but does not include the latest delta commits. Snapshot queries contain the freshest data but incur some computational overhead, which makes these queries less performant.

For more information about the tradeoffs between table and query types, see **Table & Query Types** in the Apache Hudi documentation.

**Hudi terminology change: Views are now queries**

Starting in release version 0.5.1, Apache Hudi changed some of its terminology. What were formerly views are called queries in later releases. The following table summarizes the changes between the old and new terms.

<table>
<thead>
<tr>
<th>Old term</th>
<th>New term</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoW: read optimized view</td>
<td>Snapshot queries</td>
</tr>
<tr>
<td>MoR: realtime view</td>
<td>Incremental query</td>
</tr>
<tr>
<td>Incremental view</td>
<td></td>
</tr>
<tr>
<td>MoR read optimized view</td>
<td>Read optimized query</td>
</tr>
</tbody>
</table>
Tables from bootstrap operation

Starting in Apache Hudi version 0.6.0, the bootstrap operation feature provides better performance with existing Parquet datasets. Instead of rewriting the dataset, a bootstrap operation can generate metadata only, leaving the dataset in place.

You can use Athena to query tables from a bootstrap operation just like other tables based on data in Amazon S3. In your CREATE TABLE statement, specify the Hudi table path in your LOCATION clause.

For more information about creating Hudi tables using the bootstrap operation in Amazon EMR, see the article New features from Apache Hudi available in Amazon EMR in the AWS Big Data Blog.

Considerations and limitations

- Athena does not support incremental queries.
- Athena does not support CTAS (p. 281) or INSERT INTO (p. 697) on Hudi data. If you would like Athena support for writing Hudi datasets, send feedback to <athena-feedback@amazon.com>.

For more information about writing Hudi data, see the following resources:
- Working with a Hudi dataset in the Amazon EMR Release Guide.
- Writing Data in the Apache Hudi documentation.
- Using MSCK REPAIR TABLE on Hudi tables in Athena is not supported. If you need to load a Hudi table not created in AWS Glue, use ALTER TABLE ADD PARTITION (p. 733).

Video

The following video shows how you can use Amazon Athena to query a read-optimized Apache Hudi dataset in your Amazon S3-based data lake.

Query Apache Hudi datasets using Amazon Athena

Creating Hudi tables

This section provides examples of CREATE TABLE statements in Athena for partitioned and nonpartitioned tables of Hudi data.

If you have Hudi tables already created in AWS Glue, you can query them directly in Athena. When you create partitioned Hudi tables in Athena, you must run ALTER TABLE ADD PARTITION to load the Hudi data before you can query it.

Copy on write (CoW) create table examples

Nonpartitioned CoW table

The following example creates a nonpartitioned CoW table in Athena.

```sql
CREATE EXTERNAL TABLE `non_partition_cow`(
    `_hoodie_commit_time` string,
    `_hoodie_commit_seqno` string,
    `_hoodie_record_key` string,
    `_hoodie_partition_path` string,
    `_hoodie_file_name` string,
    `event_id` string,
    `event_time` string,
```
Partitioned CoW table

The following example creates a partitioned CoW table in Athena.

```
CREATE EXTERNAL TABLE `partition_cow`(
    `_hoodie_commit_time` string,
    `_hoodie_commit_seqno` string,
    `_hoodie_record_key` string,
    `_hoodie_partition_path` string,
    `_hoodie_file_name` string,
    `event_id` string,
    `event_time` string,
    `event_name` string,
    `event_guests` int
) PARTITIONED BY (
    `event_type` string
) ROW FORMAT SERDE
    'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
STORED AS INPUTFORMAT
    'org.apache.hudi.hadoop.HoodieParquetInputFormat'
OUTPUTFORMAT
    'org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat'
LOCATION
    's3://bucket/folder/partition_cow'
```

The following ALTER TABLE ADD PARTITION example adds two partitions to the example partition_cow table.

```
ALTER TABLE partition_cow ADD
    PARTITION (event_type = 'one') LOCATION 's3://bucket/folder/partition_cow/one/'
    PARTITION (event_type = 'two') LOCATION 's3://bucket/folder/partition_cow/two/'
```

Merge on read (MoR) create table examples

Hudi creates two tables in the metastore for MoR: a table for snapshot queries, and a table for read optimized queries. Both tables are queryable. In Hudi versions prior to 0.5.1, the table for read optimized queries had the name that you specified when you created the table. Starting in Hudi version 0.5.1, the table name is suffixed with _ro by default. The name of the table for snapshot queries is the name that you specified appended with _rt.

Nonpartitioned merge on read (MoR) table

The following example creates a nonpartitioned MoR table in Athena for read optimized queries. Note that read optimized queries use the input format HoodieParquetInputFormat.

```
CREATE EXTERNAL TABLE `nonpartition_mor`(
    `_hoodie_commit_time` string,
```
Creating Hudi tables

The following example creates a nonpartitioned MoR table in Athena for snapshot queries. For snapshot queries, use the input format HoodieParquetRealtimeInputFormat.

```
CREATE EXTERNAL TABLE `nonpartition_mor_rt`
(`_hoodie_commit_time` string,
 `_hoodie_commit_seqno` string,
 `_hoodie_record_key` string,
 `_hoodie_partition_path` string,
 `_hoodie_file_name` string,
 `event_id` string,
 `event_time` string,
 `event_name` string,
 `event_guests` int,
 `event_type` string)
ROW FORMAT SERDE
  'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
STORED AS INPUTFORMAT
  'org.apache.hudi.hadoop.HoodieParquetInputFormat'
OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat'
LOCATION
  's3://bucket/folder/nonpartition_mor'
```

Partitioned merge on read (MoR) table

The following example creates a partitioned MoR table in Athena for read optimized queries.

```
CREATE EXTERNAL TABLE `partition_mor`
(`_hoodie_commit_time` string,
 `_hoodie_commit_seqno` string,
 `_hoodie_record_key` string,
 `_hoodie_partition_path` string,
 `_hoodie_file_name` string,
 `event_id` string,
 `event_time` string,
 `event_name` string,
 `event_guests` int)
PARTITIONED BY (
  `event_type` string)
ROW FORMAT SERDE
  'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
STORED AS INPUTFORMAT
  'org.apache.hudi.hadoop.HoodieParquetInputFormat'
OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat'
```

424
The following ALTER TABLE ADD PARTITION example adds two partitions to the example partition_mor table.

```
ALTER TABLE partition_mor ADD
  PARTITION (event_type = 'one') LOCATION 's3://bucket/folder/partition_mor/one/
  PARTITION (event_type = 'two') LOCATION 's3://bucket/folder/partition_mor/two/
```

The following example creates a partitioned MoR table in Athena for snapshot queries.

```
CREATE EXTERNAL TABLE `partition_mor_rt`(
  `event_id` string,
  `event_time` string,
  `event_name` string,
  `event_guests` int)
PARTITIONED BY (
  `event_type` string)
ROW FORMAT SERDE
  'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
STORED AS INPUTFORMAT
  'org.apache.hudi.hadoop.realtime.HoodieParquetRealtimeInputFormat'
OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat'
LOCATION
  's3://bucket/folder/partition_mor'
```

Similarly, the following ALTER TABLE ADD PARTITION example adds two partitions to the example partition_mor_rt table.

```
ALTER TABLE partition_mor_rt ADD
  PARTITION (event_type = 'one') LOCATION 's3://bucket/folder/partition_mor/one/
  PARTITION (event_type = 'two') LOCATION 's3://bucket/folder/partition_mor/two/
```

See also

For information about using AWS Glue custom connectors and AWS Glue 2.0 jobs to create an Apache Hudi table that you can query with Athena, see Writing to Apache Hudi tables using AWS Glue custom connector in the AWS Big Data Blog.

## Querying JSON

Amazon Athena lets you parse JSON-encoded values, extract data from JSON, search for values, and find length and size of JSON arrays.

**Topics**
- Best practices for reading JSON data (p. 426)
- Extracting data from JSON (p. 427)
- Searching for values in JSON arrays (p. 429)
Best practices for reading JSON data

JavaScript Object Notation (JSON) is a common method for encoding data structures as text. Many applications and tools output data that is JSON-encoded.

In Amazon Athena, you can create tables from external data and include the JSON-encoded data in them. For such types of source data, use Athena together with JSON SerDe libraries (p. 319).

Use the following tips to read JSON-encoded data:

• Choose the right SerDe, a native JSON SerDe, org.apache.hive.hcatalog.data.JsonSerDe, or an OpenX SerDe, org.openx.data.jsonserde.JsonSerDe. For more information, see JSON SerDe libraries (p. 319).

• Make sure that each JSON-encoded record is represented on a separate line, not pretty-printed.

  Note
  The SerDe expects each JSON document to be on a single line of text with no line termination characters separating the fields in the record. If the JSON text is in pretty print format, you may receive an error message like HIVE_CURSOR_ERROR: Row is not a valid JSON Object or HIVE_CURSOR_ERROR: JsonParseException: Unexpected end-of-input: expected close marker for OBJECT when you attempt to query the table after you create it. For more information, see JSON Data Files in the OpenX SerDe documentation on GitHub.

• Generate your JSON-encoded data in case-insensitive columns.

• Provide an option to ignore malformed records, as in this example.

```sql
CREATE EXTERNAL TABLE json_table (  
column_a string,  
column_b int  
)  
ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'  
WITH SERDEPROPERTIES ('ignore.malformed.json' = 'true')  
LOCATION 's3://bucket/path/';
```

• Convert fields in source data that have an undetermined schema to JSON-encoded strings in Athena.

When Athena creates tables backed by JSON data, it parses the data based on the existing and predefined schema. However, not all of your data may have a predefined schema. To simplify schema management in such cases, it is often useful to convert fields in source data that have an undetermined schema to JSON strings in Athena, and then use JSON SerDe libraries (p. 319).

For example, consider an IoT application that publishes events with common fields from different sensors. One of those fields must store a custom payload that is unique to the sensor sending the event. In this case, since you don't know the schema, we recommend that you store the information as a JSON-encoded string. To do this, convert data in your Athena table to JSON, as in the following example. You can also convert JSON-encoded data to Athena data types.

• Converting Athena data types to JSON (p. 426)

• Converting JSON to Athena data types (p. 427)

Converting Athena data types to JSON

To convert Athena data types to JSON, use CAST.
WITH dataset AS ( 
    SELECT 
        CAST('HELLO ATHENA' AS JSON) AS hello_msg, 
        CAST(12345 AS JSON) AS some_int, 
        CAST(MAP(ARRAY['a', 'b'], ARRAY[1,2]) AS JSON) AS some_map 
    ) 
SELECT * FROM dataset

This query returns:

<table>
<thead>
<tr>
<th>hello_msg</th>
<th>some_int</th>
<th>some_map</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;HELLO ATHENA&quot;</td>
<td>12345</td>
<td>{&quot;a&quot;:1,&quot;b&quot;:2}</td>
</tr>
</tbody>
</table>

Converting JSON to Athena data types

To convert JSON data to Athena data types, use CAST.

Note
In this example, to denote strings as JSON-encoded, start with the JSON keyword and use single quotes, such as JSON ‘12345’

WITH dataset AS ( 
    SELECT 
        CAST(JSON '"HELLO ATHENA"' AS VARCHAR) AS hello_msg, 
        CAST(JSON '12345' AS INTEGER) AS some_int, 
        CAST(JSON '{"a":1,"b":2} AS MAP(VARCHAR, INTEGER)) AS some_map 
    ) 
SELECT * FROM dataset

This query returns:

<table>
<thead>
<tr>
<th>hello_msg</th>
<th>some_int</th>
<th>some_map</th>
</tr>
</thead>
<tbody>
<tr>
<td>HELLO ATHENA</td>
<td>12345</td>
<td>{a:1,b:2}</td>
</tr>
</tbody>
</table>

Extracting data from JSON

You may have source data containing JSON-encoded strings that you do not necessarily want to deserialize into a table in Athena. In this case, you can still run SQL operations on this data, using the JSON functions available in Presto.

Consider this JSON string as an example dataset.

```json
{"name": "Susan Smith", 
"org": "engineering", 
"projects": 
    [
        {"name": "project1", "completed":false}, 
        {"name": "project2", "completed":true}
    ]
}
```
Examples: Extracting properties

To extract the name and projects properties from the JSON string, use the `json_extract` function as in the following example. The `json_extract` function takes the column containing the JSON string, and searches it using a JSONPath-like expression with the dot . notation.

- **Note**
  - JSONPath performs a simple tree traversal. It uses the $ sign to denote the root of the JSON document, followed by a period and an element nested directly under the root, such as $.name.

```
WITH dataset AS (  
    SELECT '{"name": "Susan Smith",  
            "org": "engineering",  
            "projects": ["name":"project1", "completed":false},  
                        {"name":"project2", "completed":true}]  
            AS blob  
    )  
SELECT  
    json_extract(blob, '$.name') AS name,  
    json_extract(blob, '$.projects') AS projects  
FROM dataset
```

The returned value is a JSON-encoded string, and not a native Athena data type.

```
+---------------------------+---------------------------+  
| name           | projects                  |  
|---------------------------+---------------------------+  
| "Susan Smith" | [{"name":"project1","completed":false},  
                             {"name":"project2","completed":true}] |  
+---------------------------+---------------------------+
```

To extract the scalar value from the JSON string, use the `json_extract_scalar` function. It is similar to `json_extract`, but returns only scalar values (Boolean, number, or string).

- **Note**
  - Do not use the `json_extract_scalar` function on arrays, maps, or structs.

```
WITH dataset AS (  
    SELECT '{"name": "Susan Smith",  
            "org": "engineering",  
            "projects": ["name":"project1", "completed":false},  
                        {"name":"project2", "completed":true}]  
            AS blob  
    )  
SELECT  
    json_extract_scalar(blob, '$.name') AS name,  
    json_extract_scalar(blob, '$.projects') AS projects  
FROM dataset
```

This query returns:

```
+---------------------------+---------------------------+  
| name           | projects                  |  
|---------------------------+---------------------------+  
| Susan Smith    |                          |  
+---------------------------+---------------------------+
```

428
To obtain the first element of the `projects` property in the example array, use the `json_array_get` function and specify the index position.

```
WITH dataset AS (  
    SELECT '{
    "name": "Bob Smith",
    "org": "engineering",
    "projects": [
        {
            "name": "project1",
            "completed": false
        },
        {
            "name": "project2",
            "completed": true
        }
    ]
} AS blob  
)  
SELECT json_array_get(json_extract(blob, '$.projects'), 0) AS item  
FROM dataset
```

It returns the value at the specified index position in the JSON-encoded array.

```
+------------------------------------------------------------------+
| item                                                              |
+------------------------------------------------------------------+
| {"name":"project1","completed":false}                         |
+------------------------------------------------------------------+
```

To return an Athena string type, use the `[]` operator inside a JSONPath expression, then use `json_extract_scalar` function. For more information about `[]`, see Accessing array elements (p. 389).

```
WITH dataset AS (  
    SELECT '{
    "name": "Bob Smith",
    "org": "engineering",
    "projects": [
        {
            "name": "project1",
            "completed": false
        },
        {
            "name": "project2",
            "completed": true
        }
    ]
} AS blob  
)  
SELECT json_extract_scalar(blob, '$.projects[0].name') AS project_name  
FROM dataset
```

It returns this result:

```
+------------------+
| project_name      |
+------------------+
| project1          |
+------------------+
```

### Searching for values in JSON arrays

To determine if a specific value exists inside a JSON-encoded array, use the `json_array_contains` function.

The following query lists the names of the users who are participating in "project2".

```
WITH dataset AS (  
    SELECT * FROM (VALUES
        (JSON '{
        "name": "Bob Smith",
        "org": "legal",
        "projects": ["project1"]
}'),
        (JSON '{
        "name": "Susan Smith",
        "org": "engineering",
        "projects": ["project1", "project2", "project3"]
}'),
        (JSON '{
        "name": "Jane Smith",
        "org": "finance",
        "projects": ["project1", "project2"]
}'))
    ) AS t (users)  
)  
SELECT json_extract_scalar(users, '$.name') AS user  
FROM dataset
```

429
WHERE json_array_contains(json_extract(users, '$.projects'), 'project2')

This query returns a list of users.

+-------------+
| user        |
+-------------+
| Susan Smith |
+-------------+
| Jane Smith  |

The following query example lists the names of users who have completed projects along with the total number of completed projects. It performs these actions:

• Uses nested SELECT statements for clarity.
• Extracts the array of projects.
• Converts the array to a native array of key-value pairs using CAST.
• Extracts each individual array element using the UNNEST operator.
• Filters obtained values by completed projects and counts them.

Note
When using CAST to MAP you can specify the key element as VARCHAR (native String in Presto), but leave the value as JSON, because the values in the MAP are of different types: String for the first key-value pair, and Boolean for the second.

WITH dataset AS (
    SELECT * FROM (VALUES
        (JSON '{"name": "Bob Smith",
            "org": "legal",
            "projects": [{"name":"project1", "completed":false}]'}),
        (JSON '{"name": "Susan Smith",
            "org": "engineering",
            "projects": [{"name":"project2", "completed":true},
                          {"name":"project3", "completed":true}]'}),
        (JSON '{"name": "Jane Smith",
            "org": "finance",
            "projects": [{"name":"project2", "completed":true}]'}
    ) AS t (users)
),
employees AS (
    SELECT users, CAST(json_extract(users, '$.projects') AS ARRAY(MAP(VARCHAR, JSON))) AS projects_array
    FROM dataset
),
names AS (
    SELECT json_extract_scalar(users, '$.name') AS name, projects
    FROM employees, UNNEST (projects_array) AS t(projects)
)
SELECT name, count(projects) AS completed_projects FROM names
WHERE cast(element_at(projects, 'completed') AS BOOLEAN) = true
GROUP BY name

This query returns the following result:

+---------------------+---------------------+
| name                 | completed_projects  |
+---------------------+---------------------+
| Susan Smith          | 2                   |
Obtaining length and size of JSON arrays

Example: `json_array_length`

To obtain the length of a JSON-encoded array, use the `json_array_length` function.

```
WITH dataset AS (
    SELECT * FROM (VALUES
        (JSON '{"name": "Bob Smith", "org": "legal", "projects": [{"name": "project1", "completed":false}]}'),
        (JSON '{"name": "Susan Smith", "org": "engineering", "projects": [{"name": "project2", "completed":true}, {"name": "project3", "completed":true}]}'),
        (JSON '{"name": "Jane Smith", "org": "finance", "projects": [{"name": "project2", "completed":true}]}')
    ) AS t (users)
) SELECT
    json_extract_scalar(users, '$.name') as name,
    json_array_length(json_extract(users, '$.projects')) as count
FROM dataset
ORDER BY count DESC
```

This query returns this result:

```
<table>
<thead>
<tr>
<th>name</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Susan Smith</td>
<td>2</td>
</tr>
<tr>
<td>Bob Smith</td>
<td>1</td>
</tr>
<tr>
<td>Jane Smith</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Example: `json_size`

To obtain the size of a JSON-encoded array or object, use the `json_size` function, and specify the column containing the JSON string and the JSONPath expression to the array or object.

```
WITH dataset AS (
    SELECT * FROM (VALUES
        (JSON '{"name": "Bob Smith", "org": "legal", "projects": [{"name": "project1", "completed":false}]}'),
        (JSON '{"name": "Susan Smith", "org": "engineering", "projects": [{"name": "project2", "completed":true}, {"name": "project3", "completed":true}]}'),
        (JSON '{"name": "Jane Smith", "org": "finance", "projects": [{"name": "project2", "completed":true}]}')
    ) AS t (users)
) SELECT
    json_extract_scalar(users, '$.name') as name,
    json_size(json_extract(users, '$.projects')) as count
FROM dataset
ORDER BY count DESC
```
json_extract_scalar(users, '$.name') as name,
json_size(users, '$.projects') as count
FROM dataset
ORDER BY count DESC

This query returns this result:

<table>
<thead>
<tr>
<th>name</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Susan Smith</td>
<td>2</td>
</tr>
<tr>
<td>Bob Smith</td>
<td>1</td>
</tr>
<tr>
<td>Jane Smith</td>
<td>1</td>
</tr>
</tbody>
</table>

Troubleshooting JSON queries

For help on troubleshooting issues with JSON-related queries, see JSON related errors (p. 769) or consult the following resources:

- I get errors when I try to read JSON data in Amazon Athena
- How do I resolve "HIVE_CURSOR_ERROR: Row is not a valid JSON object - JSONException: Duplicate key" when reading files from AWS Config in Athena?
- The SELECT COUNT query in Amazon Athena returns only one record even though the input JSON file has multiple records
- How can I see the Amazon S3 source file for a row in an Athena table?

See also Considerations and limitations for SQL queries in Amazon Athena (p. 764).

Using Machine Learning (ML) with Amazon Athena

Machine Learning (ML) with Amazon Athena lets you use Athena to write SQL statements that run Machine Learning (ML) inference using Amazon SageMaker. This feature simplifies access to ML models for data analysis, eliminating the need to use complex programming methods to run inference.

To use ML with Athena, you define an ML with Athena function with the USING EXTERNAL FUNCTION clause. The function points to the SageMaker model endpoint that you want to use and specifies the variable names and data types to pass to the model. Subsequent clauses in the query reference the function to pass values to the model. The model runs inference based on the values that the query passes and then returns inference results. For more information about SageMaker and how SageMaker endpoints work, see the Amazon SageMaker Developer Guide.

For an example that uses ML with Athena and SageMaker inference to detect an anomalous value in a result set, see the AWS Big Data Blog article Detecting anomalous values by invoking the Amazon Athena machine learning inference function.

Considerations and limitations

- **Available Regions** – The Athena ML feature is feature in the AWS Regions where Athena engine version 2 or later are supported.
- **SageMaker model endpoint must accept and return text/csv** – For more information about data formats, see Common data formats for inference in the Amazon SageMaker Developer Guide.
• **SageMaker endpoint scaling** – Make sure that the referenced SageMaker model endpoint is sufficiently scaled up for Athena calls to the endpoint. For more information, see [Automatically scale SageMaker models](https://docs.aws.amazon.com/sagemaker/latest/dg/automate-endpoint-scaling.html) in the Amazon SageMaker Developer Guide and [CreateEndpointConfig](https://docs.aws.amazon.com/sagemaker/latest/dg/API_CreateEndpointConfig.html) in the Amazon SageMaker API Reference.

• **IAM permissions** – To run a query that specifies an ML with Athena function, the IAM principal running the query must be allowed to perform the `sagemaker:InvokeEndpoint` action for the referenced SageMaker model endpoint. For more information, see [Allowing access for ML with Athena](https://docs.aws.amazon.com/sagemaker/latest/dg/allow-access-ml.html) (p. 570).

• **ML with Athena functions cannot be used in GROUP BY clauses directly**

## ML with Athena syntax

The **USING EXTERNAL FUNCTION** clause specifies an ML with Athena function or multiple functions that can be referenced by a subsequent SELECT statement in the query. You define the function name, variable names, and data types for the variables and return values.

### Synopsis

The following syntax shows a **USING EXTERNAL FUNCTION** clause that specifies an ML with Athena function.

```sql
USING EXTERNAL FUNCTION ml_function_name (variable1 data_type[, variable2 data_type][,...])
RETURNS data_type
SAGEMAKER 'sagemaker_endpoint'
SELECT ml_function_name()
```

### Parameters

**USING EXTERNAL FUNCTION** `ml_function_name (variable1 data_type[, variable2 data_type][,...])`

- `ml_function_name` defines the function name, which can be used in subsequent query clauses. Each `variable data_type` specifies a named variable and its corresponding data type that the SageMaker model accepts as input. The data type specified must be a supported Athena data type.

**RETURNS** `data_type`

- `data_type` specifies the SQL data type that `ml_function_name` returns to the query as output from the SageMaker model.

**SAGEMAKER 'sagemaker_endpoint'**

- `sagemaker_endpoint` specifies the endpoint of the SageMaker model.

**SELECT [...] ml_function_name(expression)[...]**

- The SELECT query that passes values to function variables and the SageMaker model to return a result. `ml_function_name` specifies the function defined earlier in the query, followed by an `expression` that is evaluated to pass values. Values that are passed and returned must match the corresponding data types specified for the function in the **USING EXTERNAL FUNCTION** clause.

### Example

The following example demonstrates a query using ML with Athena.

**Example**

```sql
USING EXTERNAL FUNCTION predict_customer_registration(age INTEGER)
```
Customer use examples

The following videos, which use the Preview version of Machine Learning (ML) with Amazon Athena, showcase ways in which you can use SageMaker with Athena.

Predicting customer churn

The following video shows how to combine Athena with the machine learning capabilities of Amazon SageMaker to predict customer churn.

Predict customer churn using Amazon Athena and Amazon SageMaker

Detecting botnets

The following video shows how one company uses Amazon Athena and Amazon SageMaker to detect botnets.

Detect botnets using Amazon Athena and Amazon SageMaker

Querying with user defined functions

User Defined Functions (UDF) in Amazon Athena allow you to create custom functions to process records or groups of records. A UDF accepts parameters, performs work, and then returns a result.

To use a UDF in Athena, you write a USING EXTERNAL FUNCTION clause before a SELECT statement in a SQL query. The SELECT statement references the UDF and defines the variables that are passed to the UDF when the query runs. The SQL query invokes a Lambda function using the Java runtime when it calls the UDF. UDFs are defined within the Lambda function as methods in a Java deployment package. Multiple UDFs can be defined in the same Java deployment package for a Lambda function. You also specify the name of the Lambda function in the USING EXTERNAL FUNCTION clause.

You have two options for deploying a Lambda function for Athena UDFs. You can deploy the function directly using Lambda, or you can use the AWS Serverless Application Repository. To find existing Lambda functions for UDFs, you can search the public AWS Serverless Application Repository or your private repository and then deploy to Lambda. You can also create or modify Java source code, package it into a JAR file, and deploy it using Lambda or the AWS Serverless Application Repository. For example Java source code and packages to get you started, see Creating and deploying a UDF using Lambda (p. 437). For more information about Lambda, see AWS Lambda Developer Guide. For more information about AWS Serverless Application Repository, see the AWS Serverless Application Repository Developer Guide.

For an example that uses UDFs with Athena to translate and analyze text, see the AWS Machine Learning Blog article Translate and analyze text using SQL functions with Amazon Athena, Amazon Translate, and Amazon Comprehend, or watch the video.

Considerations and limitations

• **Available Regions** – The Athena UDF feature is available in the AWS Regions where Athena engine version 2 or later is supported. For a list of AWS Regions that support Athena engine version 2, see Athena engine version 2 (p. 680).
• **Built-in Athena functions** – Built-in Presto functions in Athena are designed to be highly performant. We recommend that you use built-in functions over UDFs when possible. For more information about built-in functions, see Functions in Amazon Athena (p. 719).

• **Scalar UDFs only** – Athena only supports scalar UDFs, which process one row at a time and return a single column value. Athena passes a batch of rows, potentially in parallel, to the UDF each time it invokes Lambda. When designing UDFs and queries, be mindful of the potential impact to network traffic of this processing.

• **UDF handler functions use abbreviated format** – Use abbreviated format (not full format), for your UDF functions (for example, package.Class instead of package.Class::method).

• **UDF methods must be lowercase** – UDF methods must be in lowercase; camel case is not permitted.

• **Java runtime support** – Currently, Athena UDFs support the Java 8 and Java 11 runtimes for Lambda. For more information, see Building Lambda functions with Java in the AWS Lambda Developer Guide.

• **IAM permissions** – To run and create UDF query statements in Athena, the IAM principal running the query must be allowed to perform actions in addition to Athena functions. For more information, see Example IAM permissions policies to allow Amazon Athena User Defined Functions (UDF) (p. 566).

• **Lambda quotas** – Lambda quotas apply to UDFs. For more information, see Lambda quotas in the AWS Lambda Developer Guide.

• **Views** – You cannot use views with UDFs.

• **Known issues** – For the most up-to-date list of known issues, see Limitations and issues in the awslibs/aws-athena-query-federation section of GitHub.

### Videos

Watch the following videos to learn more about using UDFs in Athena.

**Video: Introducing User Defined Functions (UDFs) in Amazon Athena**

The following video shows how you can use UDFs in Amazon Athena to redact sensitive information.

**Note**

The syntax in this video is prerelease, but the concepts are the same. Use Athena engine version 2 without the AmazonAthenaPreviewFunctionality workgroup.

**Introducing user defined functions (UDFs) in Amazon Athena**

**Video: Translate, analyze, and redact text fields using SQL queries in Amazon Athena**

The following video shows how you can use UDFs in Amazon Athena together with other AWS services to translate and analyze text.

**Note**

The syntax in this video is prerelease, but the concepts are the same. For the correct syntax, see the related blog post Translate, redact, and analyze text using SQL functions with Amazon Athena, Amazon Translate, and Amazon Comprehend on the AWS Machine Learning Blog.

**Translate, analyze, and redact text fields using SQL queries in Amazon Athena**

### UDF query syntax

The **USING EXTERNAL FUNCTION** clause specifies a UDF or multiple UDFs that can be referenced by a subsequent **SELECT** statement in the query. You need the method name for the UDF and the name of the Lambda function that hosts the UDF.

**Synopsis**

```sql
USING EXTERNAL FUNCTION UDF_name(variable1 data_type[, variable2 data_type][,...])
```
UDF query syntax

<table>
<thead>
<tr>
<th>Athena data type</th>
<th>Java data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP</td>
<td>java.time.LocalDateTime (UTC)</td>
</tr>
<tr>
<td>DATE</td>
<td>java.time.LocalDate (UTC)</td>
</tr>
<tr>
<td>TINYINT</td>
<td>java.lang.Byte</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>java.lang.Short</td>
</tr>
<tr>
<td>REAL</td>
<td>java.lang.Float</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>java.lang.Double</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>BIGINT</td>
<td>java.lang.Long</td>
</tr>
<tr>
<td>INTEGER</td>
<td>java.lang.Int</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>byte[]</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>java.lang.Boolean</td>
</tr>
<tr>
<td>ARRAY</td>
<td>java.util.List</td>
</tr>
<tr>
<td>ROW</td>
<td>java.util.Map&lt;String, Object&gt;</td>
</tr>
</tbody>
</table>

**Parameters**

**USING EXTERNAL FUNCTION** `UDF_name(variable1 data_type[, variable2 data_type][,...])`  

- `UDF_name` specifies the name of the UDF, which must correspond to a Java method within the referenced Lambda function. Each `variable data_type` specifies a named variable and its corresponding data type that the UDF accepts as input. The `data_type` must be one of the supported Athena data types listed in the following table and map to the corresponding Java data type.

**RETURNS** `data_type`

- `data_type` specifies the SQL data type that the UDF returns as output. Athena data types listed in the table above are supported. For the DECIMAL data type, use the syntax `RETURNS DECIMAL(precision, scale)` where `precision` and `scale` are integers.

**LAMBDA** `'lambda_function'`

- `lambda_function` specifies the name of the Lambda function to be invoked when running the UDF.

**SELECT** `[...] UDF_name(expression) [...]`  
The SELECT query that passes values to the UDF and returns a result. `UDF_name` specifies the UDF to use, followed by an `expression` that is evaluated to pass values. Values that are passed and
Examples

For example queries based on the AthenaUDFHandler.java code on GitHub, see the GitHub Amazon Athena UDF connector page.

Creating and deploying a UDF using Lambda

To create a custom UDF, you create a new Java class by extending the UserDefinedFunctionHandler class. The source code for the UserDefinedFunctionHandler.java in the SDK is available on GitHub in the awslabs/aws-athena-query-federation/athena-federation-sdk repository, along with example UDF implementations that you can examine and modify to create a custom UDF.

The steps in this section demonstrate writing and building a custom UDF Jar file using Apache Maven from the command line and a deploy.

**Steps to Create a Custom UDF for Athena Using Maven**
- Clone the SDK and prepare your development environment (p. 437)
- Create your Maven project (p. 438)
- Add dependencies and plugins to your Maven project (p. 438)
- Write Java code for the UDFs (p. 439)
- Build the JAR file (p. 440)
- Deploy the JAR to AWS Lambda (p. 440)

Clone the SDK and prepare your development environment

Before you begin, make sure that git is installed on your system using `sudo yum install git -y`.

To install the AWS query federation SDK

- Enter the following at the command line to clone the SDK repository. This repository includes the SDK, examples and a suite of data source connectors. For more information about data source connectors, see Using Amazon Athena Federated Query (p. 60).

```
git clone https://github.com/awslabs/aws-athena-query-federation.git
```

To install prerequisites for this procedure

If you are working on a development machine that already has Apache Maven, the AWS CLI, and the AWS Serverless Application Model build tool installed, you can skip this step.

1. From the root of the aws-athena-query-federation directory that you created when you cloned, run the `prepare_dev_env.sh` script that prepares your development environment.
2. Update your shell to source new variables created by the installation process or restart your terminal session.

```
source ~/.profile
```
**Important**
If you skip this step, you will get errors later about the AWS CLI or AWS SAM build tool not being able to publish your Lambda function.

## Create your Maven project

Run the following command to create your Maven project. Replace `groupId` with the unique ID of your organization, and replace `my-athena-udf` with the name of your application. For more information, see How do I make my first Maven project? in Apache Maven documentation.

```
mvn -B archetype:generate \
-DarchetypeGroupId=org.apache.maven.archetypes \n-DgroupId=groupId \n-DartifactId=my-athena-udfs
```

## Add dependencies and plugins to your Maven project

Add the following configurations to your Maven project `pom.xml` file. For an example, see the `pom.xml` file in GitHub.

```xml
<properties>
  <aws-athena-federation-sdk.version>2021.6.1</aws-athena-federation-sdk.version>
</properties>

<dependencies>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-athena-federation-sdk</artifactId>
    <version>${aws-athena-federation-sdk.version}</version>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <version>3.2.1</version>
      <configuration>
        <createDependencyReducedPom>false</createDependencyReducedPom>
        <filters>
          <filter>
            <artifact>**</artifact>
            <excludes>
              <exclude>META-INF/*.SF</exclude>
              <exclude>META-INF/*.DSA</exclude>
              <exclude>META-INF/*.RSA</exclude>
            </excludes>
          </filter>
        </filters>
      </configuration>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>shade</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```
Write Java code for the UDFs

Create a new class by extending UserDefinedFunctionHandler.java. Write your UDFs inside the class.

In the following example, two Java methods for UDFs, compress() and decompress(), are created inside the class MyUserDefinedFunctions.

```java
package com.mycompany.athena.udfs;

public class MyUserDefinedFunctions extends UserDefinedFunctionHandler {
    private static final String SOURCE_TYPE = "MyCompany";

    public MyUserDefinedFunctions() {
        super(SOURCE_TYPE);
    }

    /**
     * Compresses a valid UTF-8 String using the zlib compression library.
     * Encodes bytes with Base64 encoding scheme.
     * @param input the String to be compressed
     * @return the compressed String
     */
    public String compress(String input) {
        byte[] inputBytes = input.getBytes(StandardCharsets.UTF_8);

        // create compressor
        Deflater compressor = new Deflater();
        compressor.setInput(inputBytes);
        compressor.finish();

        // compress bytes to output stream
        byte[] buffer = new byte[4096];
        ByteArrayOutputStream byteArrayOutputStream = new 
        ByteArrayOutputStream(inputBytes.length);
        while (!compressor.finished()) {
            int bytes = compressor.deflate(buffer);
            byteArrayOutputStream.write(buffer, 0, bytes);
        }

        try {
            byteArrayOutputStream.close();
        } catch (IOException e) {
            throw new RuntimeException("Failed to close ByteArrayOutputStream", e);
        }

        // return encoded string
        byte[] compressedBytes = byteArrayOutputStream.toByteArray();
        return Base64.getEncoder().encodeToString(compressedBytes);
    }

    /**
     * Decompresses a valid String that has been compressed using the zlib compression library.
     * Decodes bytes with Base64 decoding scheme.
     * @param input the String to be decompressed
     * @return the decompressed String
     */
    public String decompress(String input) {
        byte[] encodedBytes = Base64.getDecoder().decode(input);

        // create decompressor
        Inflater decompressor = newInflater();
        decompressor.setInput(encodedBytes);

        // decompress bytes to output stream
        byte[] buffer = new byte[4096];
        ByteArrayOutputStream byteArrayOutputStream = new 
        ByteArrayOutputStream(encodedBytes.length);
        while (!decompressor.finished()) {
            int bytes = decompressor.inflate(buffer);
            byteArrayOutputStream.write(buffer, 0, bytes);
        }

        try {
            byteArrayOutputStream.close();
        } catch (IOException e) {
            throw new RuntimeException("Failed to close ByteArrayOutputStream", e);
        }

        // return decompressed String
        byte[] decompressedBytes = byteArrayOutputStream.toByteArray();
        return new String(decompressedBytes, StandardCharsets.UTF_8);
    }
}
```
public String decompress(String input)
{
    byte[] inputBytes = Base64.getDecoder().decode(input);
    // create decompressor
    Inflater decompressor = new Inflater();
    decompressor.setInput(inputBytes, 0, inputBytes.length);
    // decompress bytes to output stream
    byte[] buffer = new byte[4096];
    ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream(inputBytes.length);
    try {
        while (!decompressor.finished()) {
            int bytes = decompressor.inflate(buffer);
            if (bytes == 0 && decompressor.needsInput()) {
                throw new DataFormatException("Input is truncated");
            }
            byteArrayOutputStream.write(buffer, 0, bytes);
        }
    } catch (DataFormatException e) {
        throw new RuntimeException("Failed to decompress string", e);
    }
    try {
        byteArrayOutputStream.close();
    } catch (IOException e) {
        throw new RuntimeException("Failed to close ByteArrayOutputStream", e);
    }
    // return decoded string
    byte[] decompressedBytes = byteArrayOutputStream.toByteArray();
    return new String(decompressedBytes, StandardCharsets.UTF_8);
}

Build the JAR file

Run mvn clean install to build your project. After it successfully builds, a JAR file is created in the target folder of your project named artifactId-version.jar, where artifactId is the name you provided in the Maven project, for example, my-athena-udfs.

Deploy the JAR to AWS Lambda

You have two options to deploy your code to Lambda:

- Deploy Using AWS Serverless Application Repository (Recommended)
- Create a Lambda Function from the JAR file

Option 1: Deploying to the AWS Serverless Application Repository

When you deploy your JAR file to the AWS Serverless Application Repository, you create an AWS SAM template YAML file that represents the architecture of your application. You then specify this YAML file and an Amazon S3 bucket where artifacts for your application are uploaded and made available to the AWS Serverless Application Repository. The procedure below uses the publish.sh script located in the
The following example demonstrates parameters in a YAML file. Add similar parameters to your YAML file and save it in your project directory. See athena-udf.yaml in GitHub for a full example.

```
Transform: 'AWS::Serverless-2016-10-31'
Metadata:
  'AWS::ServerlessRepo::Application':
    Name: MyApplicationName
    Description: 'The description I write for my application'
    Author: 'Author Name'
    Labels:
      - athena-federation
    SemanticVersion: 1.0.0
Parameters:
  LambdaFunctionName:
    Description: 'The name of the Lambda function that will contain your UDFs.'
    Type: String
  LambdaTimeout:
    Description: 'Maximum Lambda invocation runtime in seconds. (min 1 - 900 max)'
    Default: 900
    Type: Number
  LambdaMemory:
    Description: 'Lambda memory in MB (min 128 - 3008 max).'
    Default: 3008
    Type: Number
Resources:
  ConnectorConfig:
    Type: 'AWS::Serverless::Function'
    Properties:
      FunctionName: !Ref LambdaFunctionName
      Handler: "full.path.to.your.handler. For example, com.amazonaws.athena.connectors.udfs.MyUDFHandler"
      CodeUri: "Relative path to your JAR file. For example, ./target/athena-udfs-1.0.jar"
      Description: "My description of the UDFs that this Lambda function enables."
      Runtime: java8
      Timeout: !Ref LambdaTimeout
      MemorySize: !Ref LambdaMemory
```

Copy the publish.sh script to the project directory where you saved your YAML file, and run the following command:

```
./publish.sh MyS3Location MyYamlFile
```

For example, if your bucket location is s3://mybucket/mysarapps/athenaudf and your YAML file was saved as my-athena-udfs.yaml:

```
./publish.sh mybucket/mysarapps/athenaudf my-athena-udfs
```

**To create a Lambda function**

1. Open the Lambda console at https://console.aws.amazon.com/lambda/, choose **Create function**, and then choose **Browse serverless app repository**
2. Choose **Private applications**, find your application in the list, or search for it using key words, and select it.
3. Review and provide application details, and then choose **Deploy**.

You can now use the method names defined in your Lambda function JAR file as UDFs in Athena.

**Option 2: Creating a Lambda function directly**

You can also create a Lambda function directly using the console or AWS CLI. The following example demonstrates using the Lambda `create-function` CLI command.

```bash
aws lambda create-function \
--function-name MyLambdaFunctionName \
--runtime java8 \
--role arn:aws:iam::123456789012:role/my_lambda_role \
--handler com.mycompany.athena.udfs.MyUserDefinedFunctions \
--timeout 900 \
--zip-file fileb://./target/my-athena-udfs-1.0-SNAPSHOT.jar
```

**Querying across regions**

Athena supports the ability to query Amazon S3 data in an AWS Region that is different from the Region in which you are using Athena. Querying across Regions can be an option when moving the data is not practical or permissible, or if you want to query data across multiple regions. Even if Athena is not available in a particular Region, data from that Region can be queried from another Region in which Athena is available.

To query data in a Region, your account must be enabled in that Region even if the Amazon S3 data does not belong to your account. For some regions such as US East (Ohio), your access to the Region is automatically enabled when your account is created. Other Regions require that your account be “opted-in” to the Region before you can use it. For a list of Regions that require opt-in, see Available regions in the Amazon EC2 User Guide for Linux Instances. For specific instructions about opting-in to a Region, see Managing AWS regions in the Amazon Web Services General Reference.

**Considerations and limitations**

- **Data access permissions** – To successfully query Amazon S3 data from Athena across Regions, your account must have permissions to read the data. If the data that you want to query belongs to another account, the other account must grant you access to the Amazon S3 location that contains the data.
- **Data transfer charges** – Amazon S3 data transfer charges may apply for cross-region queries. For more information, see Overview of data transfer costs for common architectures in the AWS Architecture Blog.
- **AWS Glue** – You can use AWS Glue across Regions. Additional charges may apply for cross-region AWS Glue traffic. For more information, see Create cross-account and cross-region AWS Glue connections in the AWS Big Data Blog.
- **Amazon S3 encryption options** – The SSE-S3 and SSE-KMS encryption options are supported for queries across Regions; CSE-KMS is not. For more information, see Supported Amazon S3 encryption options (p. 521).

**Querying AWS Glue Data Catalog**

Because AWS Glue Data Catalog is used by many AWS services as their central metadata repository, you might want to query Data Catalog metadata. To do so, you can use SQL queries in Athena. You can use Athena to query AWS Glue catalog metadata like databases, tables, partitions, and columns.
To obtain AWS Glue Catalog metadata, you query the `information_schema` database on the Athena backend. The example queries in this topic show how to use Athena to query AWS Glue Catalog metadata for common use cases.

**Topics**
- Considerations and limitations (p. 443)
- Listing databases and searching a specified database (p. 443)
- Listing tables in a specified database and searching for a table by name (p. 444)
- Listing partitions for a specific table (p. 444)
- Listing all columns for all tables (p. 445)
- Listing or searching columns for a specified table or view (p. 445)

**Considerations and limitations**
- Instead of querying the `information_schema` database, it is possible to use individual Apache Hive DDL commands (p. 730) to extract metadata information for specific databases, tables, views, partitions, and columns from Athena. However, the output is in a non-tabular format.
- Querying `information_schema` is most performant if you have a small to moderate amount of AWS Glue metadata. If you have a large amount of metadata, errors can occur.
- You cannot use `CREATE VIEW` to create a view on the `information_schema` database.

**Listing databases and searching a specified database**

The examples in this section show how to list the databases in metadata by schema name.

**Example – Listing databases**

The following example query lists the databases from the `information_schema.schemata` table.

```sql
SELECT schema_name
FROM   information_schema.schemata
LIMIT  10;
```

The following table shows sample results.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>alb-databas1</td>
</tr>
<tr>
<td>7</td>
<td>alb_original_cust</td>
</tr>
<tr>
<td>8</td>
<td>alblogsdatabase</td>
</tr>
<tr>
<td>9</td>
<td>athena_db_test</td>
</tr>
<tr>
<td>10</td>
<td>athena_ddl_db</td>
</tr>
</tbody>
</table>

**Example – Searching a specified database**

In the following example query, rdspostgresql is a sample database.

```sql
SELECT schema_name
FROM   information_schema.schemata
```
Listing tables in a specified database and searching for a table by name

To list metadata for tables, you can query by table schema or by table name.

Example – Listing tables by schema

The following query lists tables that use the rdspostgresql table schema.

```
SELECT table_schema, table_name, table_type
FROM information_schema.tables
WHERE table_schema = 'rdspostgresql'
```

The following table shows a sample result.

<table>
<thead>
<tr>
<th>table_schema</th>
<th>table_name</th>
<th>table_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdspostgresql</td>
<td></td>
<td>rdspostgresqldb1_public_account</td>
</tr>
</tbody>
</table>

Example – Searching for a table by name

The following query obtains metadata information for the table athena1.

```
SELECT table_schema, table_name, table_type
FROM information_schema.tables
WHERE table_name = 'athena1'
```

The following table shows a sample result.

<table>
<thead>
<tr>
<th>table_schema</th>
<th>table_name</th>
<th>table_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>athena1</td>
<td>BASE TABLE</td>
</tr>
</tbody>
</table>

Listing partitions for a specific table

You can use `SHOW PARTITIONS table_name` to list the partitions for a specified table, as in the following example.
SHOW PARTITIONS cloudtrail_logs_test2

You can also use a $partitions metadata query to list the partition numbers and partition values for a specific table.

**Example – Querying the partitions for a table using the $partitions syntax**

The following example query lists the partitions for the table cloudtrail_logs_test2 using the $partitions syntax.

```
SELECT * FROM default."cloudtrail_logs_test2$partitions" ORDER BY partition_number
```

The following table shows sample results.

<table>
<thead>
<tr>
<th>table_catalog</th>
<th>table_schema</th>
<th>table_name</th>
<th>Year</th>
<th>Month</th>
<th>Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>awsdatacatalogdefault</td>
<td>cloudtrail_logs_test2</td>
<td>2020</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>awsdatacatalogdefault</td>
<td>cloudtrail_logs_test2</td>
<td>2020</td>
<td>08</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>awsdatacatalogdefault</td>
<td>cloudtrail_logs_test2</td>
<td>2020</td>
<td>08</td>
<td>12</td>
</tr>
</tbody>
</table>

**Listing all columns for all tables**

You can list all columns for all tables in AwsDataCatalog or for all tables in a specific database in AwsDataCatalog.

- To list all columns for all databases in AwsDataCatalog, use the query `SELECT * FROM information_schema.columns`.
- To restrict the results to a specific database, use `table_schema='database_name'` in the WHERE clause.

**Example – Listing all columns for all tables in a specific database**

The following example query lists all columns for all tables in the database webdata.

```
SELECT * FROM information_schema.columns WHERE table_schema = 'webdata'
```

**Listing or searching columns for a specified table or view**

You can list all columns for a table, all columns for a view, or search for a column by name in a specified database and table.

To list the columns, use a `SELECT *` query. In the FROM clause, specify `information_schema.columns`. In the WHERE clause, use `table_schema='database_name'` to specify the database and `table_name = 'table_name'` to specify the table or view that has the columns that you want to list.

**Example – Listing all columns for a specified table**

The following example query lists all columns for the table rdspostgresql1db1_public_account.
The following table shows sample results.

<table>
<thead>
<tr>
<th>table_catalog</th>
<th>table_schema</th>
<th>table_name</th>
<th>column_name</th>
<th>ordinal_position</th>
<th>is_nullable</th>
<th>data_type</th>
<th>Comment</th>
<th>extra_info</th>
</tr>
</thead>
<tbody>
<tr>
<td>awsdatacatalog</td>
<td>rdspostgresql</td>
<td>rdspostgresqldb1_public_account</td>
<td>password</td>
<td>1</td>
<td>YES</td>
<td>varchar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>awsdatacatalog</td>
<td>rdspostgresql</td>
<td>rdspostgresqldb1_public_account</td>
<td>user_id</td>
<td>2</td>
<td>YES</td>
<td>integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>awsdatacatalog</td>
<td>rdspostgresql</td>
<td>rdspostgresqldb1_public_account</td>
<td>created_on</td>
<td>3</td>
<td>YES</td>
<td>timestamp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>awsdatacatalog</td>
<td>rdspostgresql</td>
<td>rdspostgresqldb1_public_account</td>
<td>last_login</td>
<td>4</td>
<td>YES</td>
<td>timestamp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>awsdatacatalog</td>
<td>rdspostgresql</td>
<td>rdspostgresqldb1_public_account</td>
<td>email</td>
<td>5</td>
<td>YES</td>
<td>varchar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>awsdatacatalog</td>
<td>rdspostgresql</td>
<td>rdspostgresqldb1_public_account</td>
<td>username</td>
<td>6</td>
<td>YES</td>
<td>varchar</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example – Listing the columns for a specified view**

The following example query lists all the columns in the default database for the view arrayview.

```sql
SELECT *
FROM   information_schema.columns
WHERE  table_schema = 'default'
AND table_name = 'arrayview'
```

The following table shows sample results.

<table>
<thead>
<tr>
<th>table_catalog</th>
<th>table_schema</th>
<th>table_name</th>
<th>column_name</th>
<th>ordinal_position</th>
<th>is_nullable</th>
<th>data_type</th>
<th>Comment</th>
<th>extra_info</th>
</tr>
</thead>
<tbody>
<tr>
<td>awsdatacatalog</td>
<td>default</td>
<td>arrayview</td>
<td>searchdate</td>
<td>1</td>
<td>YES</td>
<td>varchar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>awsdatacatalog</td>
<td>default</td>
<td>arrayview</td>
<td>sid</td>
<td>2</td>
<td>YES</td>
<td>varchar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>awsdatacatalog</td>
<td>default</td>
<td>arrayview</td>
<td>btid</td>
<td>3</td>
<td>YES</td>
<td>varchar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>awsdatacatalog</td>
<td>default</td>
<td>arrayview</td>
<td>p</td>
<td>4</td>
<td>YES</td>
<td>varchar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>awsdatacatalog</td>
<td>default</td>
<td>arrayview</td>
<td>infantprice</td>
<td>5</td>
<td>YES</td>
<td>varchar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>awsdatacatalog</td>
<td>default</td>
<td>arrayview</td>
<td>sump</td>
<td>6</td>
<td>YES</td>
<td>varchar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>awsdatacatalog</td>
<td>default</td>
<td>arrayview</td>
<td>journeymaparray</td>
<td>7</td>
<td>YES</td>
<td>array(varchar)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example – Searching for a column by name in a specified database and table**

The following example query searches for metadata for the sid column in the arrayview view of the default database.

```sql
SELECT *
FROM   information_schema.columns
WHERE  table_schema = 'default'
```
AND table_name = 'arrayview'
AND column_name='sid'

The following table shows a sample result.

<table>
<thead>
<tr>
<th>table_catalog</th>
<th>table_schema</th>
<th>table_name</th>
<th>column_name</th>
<th>ordinal_position</th>
<th>is_nullable</th>
<th>data_type</th>
<th>Comment</th>
<th>extra_info</th>
</tr>
</thead>
<tbody>
<tr>
<td>awsdatacatalog</td>
<td>default</td>
<td>arrayview</td>
<td>sid</td>
<td>2</td>
<td>YES</td>
<td>varchar</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Querying AWS service logs

This section includes several procedures for using Amazon Athena to query popular datasets, such as AWS CloudTrail logs, Amazon CloudFront logs, Classic Load Balancer logs, Application Load Balancer logs, Amazon VPC flow logs, and Network Load Balancer logs.

The tasks in this section use the Athena console, but you can also use other tools like the Athena JDBC driver (p. 183), the AWS CLI, or the Amazon Athena API Reference.

For information about using AWS CloudFormation to automatically create AWS service log tables, partitions, and example queries in Athena, see Automating AWS service logs table creation and querying them with Amazon Athena in the AWS Big Data Blog. For information about using a Python library for AWS Glue to create a common framework for processing AWS service logs and querying them in Athena, see Easily query AWS service logs using Amazon Athena.

The topics in this section assume that you have set up both an IAM user with appropriate permissions to access Athena and the Amazon S3 bucket where the data to query should reside. For more information, see Setting up (p. 7) and Getting started (p. 9).

**Topics**

- Querying Application Load Balancer logs (p. 447)
- Querying Classic Load Balancer logs (p. 451)
- Querying Amazon CloudFront logs (p. 452)
- Querying AWS CloudTrail logs (p. 454)
- Querying Amazon EMR logs (p. 462)
- Querying AWS Global Accelerator flow logs (p. 465)
- Querying Amazon GuardDuty findings (p. 467)
- Querying AWS Network Firewall logs (p. 469)
- Querying Network Load Balancer logs (p. 470)
- Querying Amazon Route 53 resolver query logs (p. 472)
- Querying Amazon SES event logs (p. 474)
- Querying Amazon VPC flow logs (p. 474)
- Querying AWS WAF logs (p. 481)

### Querying Application Load Balancer logs

An Application Load Balancer is a load balancing option for Elastic Load Balancing that enables traffic distribution in a microservices deployment using containers. Querying Application Load Balancer logs allows you to see the source of traffic, latency, and bytes transferred to and from Elastic Load Balancing...
instances and backend applications. For more information, see Access logs for your Application Load Balancer in the User Guide for Application Load Balancers.

Topics

- Prerequisites (p. 448)
- Creating the table for ALB logs (p. 448)
- Creating the table for ALB logs in Athena using partition projection (p. 449)
- Example queries for ALB logs (p. 450)

Prerequisites

- Enable access logging so that Application Load Balancer logs can be saved to your Amazon S3 bucket.
- A database to hold the table that you will create for Athena. To create a database, you can use the Athena or AWS Glue console. For more information, see Creating databases in Athena (p. 251) in this guide or Working with databases on the AWS glue console in the AWS Glue Developer Guide.

Creating the table for ALB logs

1. Copy and paste the following CREATE TABLE statement into the query editor in the Athena console. For information about getting started with the Athena console, see Getting started (p. 9). Replace the values in LOCATION 's3://your-alb-logs-directory/AWSSLogs/<ACCOUNT-ID>/elasticloadbalancing/<REGION>' with those corresponding to your Amazon S3 bucket location. For information about each field, see Access log entries in the User Guide for Application Load Balancers.

   **Note**
   The following CREATE TABLE statement includes the recently added classification and classification_reason columns. To create a table for Application Load Balancer access logs that do not contain these entries, remove these two columns from the CREATE TABLE statement and modify the regex accordingly.

   ```sql
   CREATE EXTERNAL TABLE IF NOT EXISTS alb_logs (
     type string,
     time string,
     elb string,
     client_ip string,
     client_port int,
     target_ip string,
     target_port int,
     request_processing_time double,
     target_processing_time double,
     response_processing_time double,
     elb_status_code int,
     target_status_code string,
     received_bytes bigint,
     sent_bytes bigint,
     request_verb string,
     request_url string,
     request_proto string,
     user_agent string,
     ssl_cipher string,
     ssl_protocol string,
     target_group_arn string,
     trace_id string,
     domain_name string,
     chosen_cert_arn string,
     matched_rule_priority string,
   )
   ```
2. Run the query in the Athena console. After the query completes, Athena registers the alb_logs table, making the data in it ready for you to issue queries.

Creating the table for ALB logs in Athena using partition projection

Because ALB logs have a known structure whose partition scheme you can specify in advance, you can reduce query runtime and automate partition management by using the Athena partition projection feature. Partition projection automatically adds new partitions as new data is added. This removes the need for you to manually add partitions by using ALTER TABLE ADD PARTITION.

The following example CREATE TABLE statement automatically uses partition projection on ALB logs from a specified date until the present for a single AWS region. The statement is based on the example in the previous section but adds PARTITIONED BY and TBLPROPERTIES clauses to enable partition projection. In the LOCATION and storage.location.template clauses, replace the placeholders with values that identify the Amazon S3 bucket location of your ALB logs. For projection.day.range, replace 2022/01/01 with the starting date that you want to use. After you run the query successfully, you can query the table. You do not have to run ALTER TABLE ADD PARTITION to load the partitions.

```
CREATE EXTERNAL TABLE IF NOT EXISTS alb_logs (  
type string,  
time string,  
elb string,  
client_ip string,  
client_port int,  
target_ip string,  
target_port int,  
request_processing_time double,  
target_processing_time double,  
response_processing_time double,  
 elb_status_code int,  
target_status_code string,  
 received_bytes bigint,  
sent_bytes bigint,  
 request_verb string,  
 request_url string,  
 request_proto string,  
 user_agent string,  
 ssl_cipher string,  
 ssl_protocol string,
```
Example queries for ALB logs

The following query counts the number of HTTP GET requests received by the load balancer grouped by the client IP address:

```sql
SELECT COUNT(request_verb) AS count, request_verb, client_ip
FROM alb_logs
GROUP BY request_verb, client_ip
LIMIT 100;
```

Another query shows the URLs visited by Safari browser users:

```sql
SELECT request_url
FROM alb_logs
WHERE user_agent LIKE 'Safari'
LIMIT 10;
```
The following query shows records that have ELB status code values greater than or equal to 500.

```
SELECT * FROM alb_logs
WHERE elb_status_code >= 500
```

The following example shows how to parse the logs by datetime:

```
SELECT client_ip, sum(received_bytes)
FROM alb_logs
WHERE parse_datetime(time,'yyyy-MM-dd''T''HH:mm:ss.SSSSSS''Z')
    BETWEEN parse_datetime('2018-05-30-12:00:00','yyyy-MM-dd-HH:mm:ss')
    AND parse_datetime('2018-05-31-00:00:00','yyyy-MM-dd-HH:mm:ss')
GROUP BY client_ip;
```

The following query queries the table that uses partition projection for all ALB logs from the specified day.

```
SELECT *
FROM alb_logs
WHERE day = '2022/02/12'
```

- For more information and examples, see the AWS Knowledge Center article How do I analyze my Application Load Balancer access logs using Athena?.
- For information about Elastic Load Balancing HTTP status codes, see Troubleshoot your application load balancers in the User Guide for Application Load Balancers.

### Querying Classic Load Balancer logs

Use Classic Load Balancer logs to analyze and understand traffic patterns to and from Elastic Load Balancing instances and backend applications. You can see the source of traffic, latency, and bytes that have been transferred.

Before you analyze the Elastic Load Balancing logs, configure them for saving in the destination Amazon S3 bucket. For more information, see Enable access logs for your Classic Load Balancer.

- Create the table for Elastic Load Balancing logs (p. 451)
- Elastic Load Balancing example queries (p. 452)

### To create the table for Elastic Load Balancing logs

1. Copy and paste the following DDL statement into the Athena console. Check the syntax of the Elastic Load Balancing log records. You may need to update the following query to include the columns and the Regex syntax for latest version of the record.

```
CREATE EXTERNAL TABLE IF NOT EXISTS elb_logs (
    timestamp string,
    elb_name string,
    request_ip string,
    request_port int,
    backend_ip string,
    backend_port int,
    request_processing_time double,
    backend_processing_time double,
    client_response_time double,
    ...
)
```
elb_response_code string,
backend_response_code string,
received_bytes bigint,
sent_bytes bigint,
request_verb string,
url string,
protocol string,
user_agent string,
ssl_cipher string,
ssl_protocol string
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
'serialization.format' = '1',
'input.regex' = '([^ ]*) ([^ ]*) ([^ ]*):([0-9]*) ([^ ]*)([-][0-9]*) ([^ ]*)
([-][0-9]*) ([^ ]*) ([0-9]*) ([^ ]*) \"([^ ]*) (\") (\") (\") (\") ([A-Za-z-0-9]*$'
)
LOCATION 's3://your_log_bucket/prefix/AWSLogs/AWS_account_ID/elasticloadbalancing/';

2. Modify the LOCATION Amazon S3 bucket to specify the destination of your Elastic Load Balancing logs.

3. Run the query in the Athena console. After the query completes, Athena registers the elb_logs table, making the data in it ready for queries. For more information, see Elastic Load Balancing example queries (p. 452).

### Elastic Load Balancing example queries

Use a query similar to the following example. It lists the backend application servers that returned a 4XX or 5XX error response code. Use the LIMIT operator to limit the number of logs to query at a time.

```
SELECT
  timestamp,
  elb_name,
  backend_ip,
  backend_response_code
FROM elb_logs
WHERE backend_response_code LIKE '4%' OR
  backend_response_code LIKE '5%'
LIMIT 100;
```

Use a subsequent query to sum up the response time of all the transactions grouped by the backend IP address and Elastic Load Balancing instance name.

```
SELECT sum(backend_processing_time) AS total_ms,
  elb_name,
  backend_ip
FROM elb_logs
WHERE backend_ip <> ''
GROUP BY backend_ip, elb_name
LIMIT 100;
```

For more information, see Analyzing data in S3 using Athena.

### Querying Amazon CloudFront logs

You can configure Amazon CloudFront CDN to export Web distribution access logs to Amazon Simple Storage Service. Use these logs to explore users’ surfing patterns across your web properties served by CloudFront.
Before you begin querying the logs, enable Web distributions access log on your preferred CloudFront distribution. For information, see Access logs in the Amazon CloudFront Developer Guide.

Make a note of the Amazon S3 bucket to which to save these logs.

**Note**
This procedure works for the Web distribution access logs in CloudFront. It does not apply to streaming logs from RTMP distributions.

- Creating the table for CloudFront logs (p. 453)
- Example query for CloudFront logs (p. 454)

### Creating the table for CloudFront logs

To create the CloudFront table

1. Copy and paste the following DDL statement into the Query Editor in the Athena console. Modify the LOCATION for the Amazon S3 bucket that stores your logs. For information about using the Query Editor, see Getting started (p. 9).

   This query uses the default SerDe, LazySimpleSerDe (p. 324). The column date is escaped using backticks (`) because it is a reserved word in Athena. For information, see Reserved keywords (p. 256).

   ```sql
   CREATE EXTERNAL TABLE IF NOT EXISTS default.cloudfront_logs (  
     `date` DATE,  
     time STRING,  
     location STRING,  
     bytes BIGINT,  
     request_ip STRING,  
     method STRING,  
     host STRING,  
     uri STRING,  
     status INT,  
     referrer STRING,  
     user_agent STRING,  
     query_string STRING,  
     cookie STRING,  
     result_type STRING,  
     request_id STRING,  
     host_header STRING,  
     request_protocol STRING,  
     request_bytes BIGINT,  
     time_taken FLOAT,  
     xforwarded_for STRING,  
     ssl_protocol STRING,  
     ssl_cipher STRING,  
     response_result_type STRING,  
     http_version STRING,  
     fle_status STRING,  
     fle_encrypted_fields INT,  
     c_port INT,  
     time_to_first_byte FLOAT,  
     x_edge_detailed_result_type STRING,  
     sc_content_type STRING,  
     sc_content_len BIGINT,  
     sc_range_start BIGINT,  
     sc_range_end BIGINT  
   )  
   ROW FORMAT DELIMITED  
   FIELDS TERMINATED BY '\t'  
   LOCATION 's3://CloudFront_bucket_name/'
   ```
TBLPROPERTIES ( 'skip.header.line.count'='2' )

2. Run the query in Athena console. After the query completes, Athena registers the `cloudfront_logs` table, making the data in it ready for you to issue queries.

Example query for CloudFront logs

The following query adds up the number of bytes served by CloudFront between June 9 and June 11, 2018. Surround the date column name with double quotes because it is a reserved word.

```sql
SELECT SUM(bytes) AS total_bytes
FROM cloudfront_logs
WHERE "date" BETWEEN DATE '2018-06-09' AND DATE '2018-06-11'
LIMIT 100;
```

To eliminate duplicate rows (for example, duplicate empty rows) from the query results, you can use the `SELECT DISTINCT` statement, as in the following example.

```sql
SELECT DISTINCT *
FROM cloudfront_logs
LIMIT 10;
```

Additional resources

For more information about using Athena to query CloudFront logs, see the following posts from the AWS big data blog.

- Easily query AWS service logs using Amazon Athena (May 29, 2019).
- Analyze your Amazon CloudFront access logs at scale (December 21, 2018).
- Build a serverless architecture to analyze Amazon CloudFront access logs using AWS Lambda, Amazon Athena, and Amazon Kinesis Data Analytics (May 26, 2017).

Querying AWS CloudTrail logs

AWS CloudTrail is a service that records AWS API calls and events for Amazon Web Services accounts.

CloudTrail logs include details about any API calls made to your AWS services, including the console. CloudTrail generates encrypted log files and stores them in Amazon S3. For more information, see the AWS CloudTrail User Guide.

**Note**

If you want to perform SQL queries on CloudTrail event information across accounts, regions, and dates, consider using CloudTrail Lake. CloudTrail Lake is an AWS alternative to creating trails that aggregates information from an enterprise into a single, searchable event data store. Instead of using Amazon S3 bucket storage, it stores events in a data lake, which allows richer, faster queries. You can use it to create SQL queries that search events across organizations, regions, and within custom time ranges. Because you perform CloudTrail Lake queries within the CloudTrail console itself, using CloudTrail Lake does not require Athena. For more information, see the CloudTrail Lake documentation.

Using Athena with CloudTrail logs is a powerful way to enhance your analysis of AWS service activity. For example, you can use queries to identify trends and further isolate activity by attributes, such as source IP address or user.
A common application is to use CloudTrail logs to analyze operational activity for security and compliance. For information about a detailed example, see the AWS Big Data Blog post, *Analyze security, compliance, and operational activity using AWS CloudTrail and Amazon Athena*.

You can use Athena to query these log files directly from Amazon S3, specifying the LOCATION of log files. You can do this one of two ways:

- By creating tables for CloudTrail log files directly from the CloudTrail console.
- By manually creating tables for CloudTrail log files in the Athena console.

**Topics**

- Understanding CloudTrail logs and Athena tables (p. 455)
- Using the CloudTrail console to create an Athena table for CloudTrail logs (p. 456)
- Creating a table for CloudTrail logs in Athena using manual partitioning (p. 456)
- Creating a table for an organization wide trail using manual partitioning (p. 458)
- Creating the table for CloudTrail logs in Athena using partition projection (p. 459)
- Querying nested fields (p. 460)
- Example query (p. 461)
- Tips for querying CloudTrail logs (p. 461)

**Understanding CloudTrail logs and Athena tables**

Before you begin creating tables, you should understand a little more about CloudTrail and how it stores data. This can help you create the tables that you need, whether you create them from the CloudTrail console or from Athena.

CloudTrail saves logs as JSON text files in compressed gzip format (*.json.gzip). The location of the log files depends on how you set up trails, the AWS Region or Regions in which you are logging, and other factors.

For more information about where logs are stored, the JSON structure, and the record file contents, see the following topics in the AWS CloudTrail User Guide:

- Finding your CloudTrail log files
- CloudTrail Log File examples
- CloudTrail record contents
- CloudTrail event reference

To collect logs and save them to Amazon S3, enable CloudTrail from the AWS Management Console. For more information, see *Creating a trail* in the AWS CloudTrail User Guide.

Note the destination Amazon S3 bucket where you save the logs. Replace the LOCATION clause with the path to the CloudTrail log location and the set of objects with which to work. The example uses a LOCATION value of logs for a particular account, but you can use the degree of specificity that suits your application.

For example:

- To analyze data from multiple accounts, you can roll back the LOCATION specifier to indicate all AWSLogs by using LOCATION 's3://MyLogFiles/AWSLogs/'.
- To analyze data from a specific date, account, and Region, use LOCATION 's3://MyLogFiles/123456789012/CloudTrail/us-east-1/2016/03/14/'.

455
Using the highest level in the object hierarchy gives you the greatest flexibility when you query using Athena.

**Using the CloudTrail console to create an Athena table for CloudTrail logs**

You can create a non-partitioned Athena table for querying CloudTrail logs directly from the CloudTrail console. Creating an Athena table from the CloudTrail console requires that you be logged in with an IAM user or role that has sufficient permissions to create tables in Athena.

*Note*
You cannot use the CloudTrail console to create an Athena table for organization trail logs. Instead, create the table manually using the Athena console so that you can specify the correct storage location. For information about organization trails, see Creating a trail for an organization in the *AWS CloudTrail User Guide*.

- For information about setting up permissions for Athena, see Setting up (p. 7).
- For information about creating a table with partitions, see Creating a table for CloudTrail logs in Athena using manual partitioning.

**To create an Athena table for a CloudTrail trail using the CloudTrail console**

1. Open the CloudTrail console at https://console.aws.amazon.com/cloudtrail/.
2. In the navigation pane, choose Event history.
3. Choose Create Athena table.

4. For Storage location, use the down arrow to select the Amazon S3 bucket where log files are stored for the trail to query.

   *Note*
   To find the name of the bucket that is associated with a trail, choose Trails in the CloudTrail navigation pane and view the trail's S3 bucket column. To see the Amazon S3 location for the bucket, choose the link for the bucket in the S3 bucket column. This opens the Amazon S3 console to the CloudTrail bucket location.

5. Choose Create table. The table is created with a default name that includes the name of the Amazon S3 bucket.

**Creating a table for CloudTrail logs in Athena using manual partitioning**

You can manually create tables for CloudTrail log files in the Athena console, and then run queries in Athena.

**To create an Athena table for a CloudTrail trail using the Athena console**

1. Copy and paste the following DDL statement into the Athena console query editor.

   ```sql
   CREATE EXTERNAL TABLE cloudtrail_logs (  
   ```
2. (Optional) Remove any fields not required for your table. If you need to read only a certain set of columns, your table definition can exclude the other columns.

3. Modify s3://CloudTrail_bucket_name/AWSLogs/Account_ID/CloudTrail/ to point to the Amazon S3 bucket that contains your log data.

4. Verify that fields are listed correctly. For more information about the full list of fields in a CloudTrail record, see CloudTrail record contents.
The following example uses the Hive JSON SerDe (p. 320). In this example, the fields requestparameters, responseelements, and additionaleventdata are listed as type STRING in the query, but are STRUCT data type used in JSON. Therefore, to get data out of these fields, use JSON_EXTRACT functions. For more information, see the section called “Extracting data from JSON” (p. 427). For performance improvements, the example partitions the data by AWS Region, year, month, and day.

5. Run the CREATE TABLE statement in the Athena console.
6. Use the ALTER TABLE ADD PARTITION (p. 733) command to load the partitions so that you can query them, as in the following example.

```
ALTER TABLE table_name ADD
PARTITION (region='us-east-1',
year='2019',
month='02',
day='01')
LOCATION 's3://cloudtrail_bucket_name/AWSLogs/Account_ID/CloudTrail/us-east-1/2019/02/01/
```

Creating a table for an organization wide trail using manual partitioning

To create a table for organization wide CloudTrail log files in Athena, follow the steps in Creating a table for CloudTrail logs in Athena using manual partitioning (p. 456), but make the modifications noted in the following procedure.

To create an Athena table for organization wide CloudTrail logs

1. In the CREATE TABLE statement, modify the LOCATION clause to include the organization ID instead of the account ID, as in the following example:

```
LOCATION 's3://cloudtrail_bucket_name/AWSLogs/organization_id/CloudTrail/
```

2. In the PARTITIONED BY clause, add an entry for the account ID as a string, as in the following example:

```
PARTITIONED BY (account string, region string, year string, month string, day string)
```

The following example shows the combined result:

```
...
PARTITIONED BY (account string, region string, year string, month string, day string)
ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe'
STORED AS INPUTFORMAT 'com.amazon.emr.cloudtrail.CloudTrailInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION 's3://cloudtrail_bucket_name/AWSLogs/organization_id/CloudTrail/
```

3. In the ALTER TABLE statement ADD PARTITION clause, include the account ID, as in the following example:

```
ALTER TABLE table_name ADD
PARTITION (account='111122223333',
region='us-east-1',
year='2022',
month='08',
```
4. In the ALTER TABLE statement LOCATION clause, include the organization ID, the account ID, and the partition that you want to add, as in the following example:

```
LOCATION 's3://cloudtrail_bucket_name/AWSLogs/organization_id/account_id/CloudTrail/us-east-1/2022/08/08/
```

The following example ALTER TABLE statement shows the combined result:

```
ALTER TABLE table_name ADD PARTITION (account='111122223333',
region='us-east-1',
year='2022',
month='08',
day='08')
LOCATION 's3://cloudtrail_bucket_name/AWSLogs/organization_id/111122223333/CloudTrail/us-east-1/2022/08/08/
```

Creating the table for CloudTrail logs in Athena using partition projection

Because CloudTrail logs have a known structure whose partition scheme you can specify in advance, you can reduce query runtime and automate partition management by using the Athena partition projection feature. Partition projection automatically adds new partitions as new data is added. This removes the need for you to manually add partitions by using ALTER TABLE ADD PARTITION.

The following example CREATE TABLE statement automatically uses partition projection on CloudTrail logs from a specified date until the present for a single AWS Region. In the LOCATION and storage.location.template clauses, replace the bucket, account-id, and aws-region placeholders with correspondingly identical values. For projection.timestamp.range, replace 2020/01/01 with the starting date that you want to use. After you run the query successfully, you can query the table. You do not have to run ALTER TABLE ADD PARTITION to load the partitions.

```
CREATE EXTERNAL TABLE cloudtrail_logs_pp(
    eventVersion STRING,
    userIdentity STRUCT<
        type: STRING,
        principalId: STRING,
        arn: STRING,
        accountId: STRING,
        invokedBy: STRING,
        accessKeyId: STRING,
        userName: STRING,
        sessionContext: STRUCT<
            attributes: STRUCT<
                mfaAuthenticated: STRING,
                creationDate: STRING>,
            sessionIssuer: STRUCT<
                type: STRING,
                principalId: STRING,
                arn: STRING,
                accountId: STRING,
                userName: STRING>,
            ec2RoleDelivery:string,
            webIdFederationData:map<string,string>
        >
    >,
);
eventTime STRING,
eventSource STRING,
eventName STRING,
awsRegion STRING,
sourceIpAddress STRING,
userAgent STRING,
errorCode STRING,
errorMessage STRING,
requestParameters STRING,
responseElements STRING,
additionalEventData STRING,
requestId STRING,
eventId STRING,
readOnly STRING,
resources ARRAY<STRUCT<
    arn: STRING,
    accountId: STRING,
    type: STRING>>,
eventType STRING,
apiVersion STRING,
recipientAccountId STRING,
serviceEventDetails STRING,
sharedEventID STRING,
vpcEndpointId STRING,
tlsDetails struct<
    tlsVersion:string,
    cipherSuite:string,
    clientProvidedHostHeader:string>
 />
PARTITIONED BY (  
    `timestamp` string)
ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe'
STORED AS INPUTFORMAT 'com.amazon.emr.cloudtrail.CloudTrailInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION  
's3://bucket/AWSLogs/account-id/CloudTrail/aws-region'
TBLPROPERTIES (  
    'projection.enabled'='true',
    'projection.timestamp.format'='yyyy/MM/dd',
    'projection.timestamp.interval'='1',
    'projection.timestamp.interval.unit'='DAYS',
    'projection.timestamp.range'='2020/01/01,NOW',
    'projection.timestamp.type'='date',
    'storage.location.template'='s3://bucket/AWSLogs/account-id/CloudTrail/aws-region/${timestamp}')

For more information about partition projection, see Partition projection with Amazon Athena (p. 265).

**Querying nested fields**

Because the userIdentity and resources fields are nested data types, querying them requires special treatment.

The userIdentity object consists of nested STRUCT types. These can be queried using a dot to separate the fields, as in the following example:

```sql
SELECT  
eventSource,
eventName,
userIdentity.sessionContext.attributes.creationDate,
userIdentity.sessionContext.sessionIssuer.arn
FROM cloudtrail_logs
WHERE userIdentity.sessionContext.sessionIssuer.arn IS NOT NULL
ORDER BY eventSource, eventName
```
The `resources` field is an array of STRUCT objects. For these arrays, use `CROSS JOIN UNNEST` to unnest the array so that you can query its objects.

The following example returns all rows where the resource ARN ends in `example/datafile.txt`. For readability, the `replace` function removes the initial `arn:aws:s3:::` substring from the ARN.

```
SELECT awsregion,
    replace(unnested.resources_entry.ARN,'arn:aws:s3:::') as s3_resource,
    eventname,
    eventtime,
    useragent
FROM cloudtrail_logs t
CROSS JOIN UNNEST(t.resources) unnested (resources_entry)
WHERE unnested.resources_entry.ARN LIKE '%example/datafile.txt'
ORDER BY eventtime
```

The following example queries for `DeleteBucket` events. The query extracts the name of the bucket and the account ID to which the bucket belongs from the `resources` object.

```
SELECT awsregion,
    replace(unnested.resources_entry.ARN,'arn:aws:s3:::') as deleted_bucket,
    eventtime AS time_deleted,
    useridentity.username,
    unnested.resources_entry.accountid as bucket_acct_id
FROM cloudtrail_logs t
CROSS JOIN UNNEST(t.resources) unnested (resources_entry)
WHERE eventname = 'DeleteBucket'
ORDER BY eventtime
```

For more information about unnesting, see Filtering arrays (p. 393).

**Example query**

The following example shows a portion of a query that returns all anonymous (unsigned) requests from the table created for CloudTrail event logs. This query selects those requests where `useridentity.accountid` is anonymous, and `useridentity.arn` is not specified:

```
SELECT *
FROM cloudtrail_logs
WHERE
eventsource = 's3.amazonaws.com' AND
    eventname in ('GetObject') AND
    useridentity.accountid = 'anonymous' AND
    useridentity.arn IS NULL AND
    requestparameters LIKE '%[your bucket name ]%';
```

For more information, see the AWS Big Data blog post Analyze security, compliance, and operational activity using AWS CloudTrail and Amazon Athena.

**Tips for querying CloudTrail logs**

To explore the CloudTrail logs data, use these tips:

- Before querying the logs, verify that your logs table looks the same as the one in the section called “Creating a table for CloudTrail logs in Athena using manual partitioning” (p. 456). If it is not the first table, delete the existing table using the following command: `DROP TABLE cloudtrail_logs`.
After you drop the existing table, re-create it. For more information, see Creating a table for CloudTrail logs in Athena using manual partitioning (p. 456).

Verify that fields in your Athena query are listed correctly. For information about the full list of fields in a CloudTrail record, see CloudTrail record contents.

If your query includes fields in JSON formats, such as STRUCT, extract data from JSON. For more information, see Extracting data from JSON (p. 427).

Some suggestions for issuing queries against your CloudTrail table:
• Start by looking at which IAM users called which API operations and from which source IP addresses.
• Use the following basic SQL query as your template. Paste the query to the Athena console and run it.

```sql
SELECT
  useridentity.arn,
  eventname,
  sourceipaddress,
  eventtime
FROM cloudtrail_logs
LIMIT 100;
```

• Modify the query to further explore your data.
• To improve performance, include the LIMIT clause to return a specified subset of rows.

### Querying Amazon EMR logs

Amazon EMR and big data applications that run on Amazon EMR produce log files. Logs files are written to the master node, and you can also configure Amazon EMR to archive log files to Amazon S3 automatically. You can use Amazon Athena to query these logs to identify events and trends for applications and clusters. For more information about the types of log files in Amazon EMR and saving them to Amazon S3, see View log files in the Amazon EMR Management Guide.

#### Creating and querying a basic table based on Amazon EMR log files

The following example creates a basic table, `myemrlogs`, based on log files saved to `s3://aws-logs-123456789012-us-west-2/elasticmapreduce/j-2ABCDE34F5GH6/elasticmapreduce/`. The Amazon S3 location used in the examples below reflects the pattern of the default log location for an EMR cluster created by Amazon Web Services account `123456789012` in Region `us-west-2`. If you use a custom location, the pattern is `s3://PathToEMRLogs/ClusterID`.

For information about creating a partitioned table to potentially improve query performance and reduce data transfer, see Creating and querying a partitioned table based on Amazon EMR logs (p. 463).

```sql
CREATE EXTERNAL TABLE `myemrlogs`(
  `data` string COMMENT 'from deserializer')
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|' LINES TERMINATED BY '\n'
STORED AS INPUTFORMAT
  'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION
  's3://aws-logs-123456789012-us-west-2/elasticmapreduce/j-2ABCDE34F5GH6'
```

The following example queries can be run on the `myemrlogs` table created by the previous example.
### Example – Query step logs for occurrences of ERROR, WARN, INFO, EXCEPTION, FATAL, or DEBUG

```sql
SELECT data, "$PATH"
FROM "default"."myemrlogs"
WHERE regexp_like("$PATH", 's-86URH188Z6B1')
    AND regexp_like(data, 'ERROR|WARN|INFO|EXCEPTION|FATAL|DEBUG') limit 100;
```

### Example – Query a specific instance log, i-00b3c0a839ece0a9c, for ERROR, WARN, INFO, EXCEPTION, FATAL, or DEBUG

```sql
SELECT "data", "$PATH" AS filepath
FROM "default"."myemrlogs"
WHERE regexp_like("$PATH", 'i-00b3c0a839ece0a9c')
    AND regexp_like("$PATH", 'state')
    AND regexp_like(data, 'ERROR|WARN|INFO|EXCEPTION|FATAL|DEBUG') limit 100;
```

### Example – Query presto application logs for ERROR, WARN, INFO, EXCEPTION, FATAL, or DEBUG

```sql
SELECT "data", "$PATH" AS filepath
FROM "default"."myemrlogs"
WHERE regexp_like("$PATH", 'presto')
    AND regexp_like(data, 'ERROR|WARN|INFO|EXCEPTION|FATAL|DEBUG') limit 100;
```

### Example – Query Namenode application logs for ERROR, WARN, INFO, EXCEPTION, FATAL, or DEBUG

```sql
SELECT "data", "$PATH" AS filepath
FROM "default"."myemrlogs"
WHERE regexp_like("$PATH", 'namenode')
    AND regexp_like(data, 'ERROR|WARN|INFO|EXCEPTION|FATAL|DEBUG') limit 100;
```

### Example – Query all logs by date and hour for ERROR, WARN, INFO, EXCEPTION, FATAL, or DEBUG

```sql
SELECT distinct("$PATH") AS filepath
FROM "default"."myemrlogs"
WHERE regexp_like("$PATH", '2019-07-23-10')
    AND regexp_like(data, 'ERROR|WARN|INFO|EXCEPTION|FATAL|DEBUG') limit 100;
```

## Creating and querying a partitioned table based on Amazon EMR logs

These examples use the same log location to create an Athena table, but the table is partitioned, and a partition is then created for each log location. For more information, see Partitioning data in Athena (p. 260).

The following query creates the partitioned table named `mypartitionedemrlogs`:

```sql
CREATE EXTERNAL TABLE `mypartitionedemrlogs`
(
    'data' string COMMENT 'from deserializer'
) partitioned by (logtype string)
```
The following query statements then create table partitions based on sub-directories for different log types that Amazon EMR creates in Amazon S3:

```sql
ALTER TABLE mypartitionedemrlogs ADD
  PARTITION (logtype='containers')
```

```sql
ALTER TABLE mypartitionedemrlogs ADD
  PARTITION (logtype='hadoop-mapreduce')
```

```sql
ALTER TABLE mypartitionedemrlogs ADD
  PARTITION (logtype='hadoop-state-pusher')
```

```sql
ALTER TABLE mypartitionedemrlogs ADD
  PARTITION (logtype='node')
  LOCATION 's3://aws-logs-123456789012-us-west-2/elasticmapreduce/j-2ABCDE34F5GH6/node/'
```

```sql
ALTER TABLE mypartitionedemrlogs ADD
  PARTITION (logtype='steps')
  LOCATION 's3://aws-logs-123456789012-us-west-2/elasticmapreduce/j-2ABCDE34F5GH6/steps/'
```

After you create the partitions, you can run a `SHOW PARTITIONS` query on the table to confirm:

```sql
SHOW PARTITIONS mypartitionedemrlogs;
```

The following examples demonstrate queries for specific log entries use the table and partitions created by the examples above.

**Example – Querying application application_1561661818238_0002 logs in the containers partition for ERROR or WARN**

```sql
SELECT data, "$PATH"
FROM "default"."mypartitionedemrlogs"
WHERE logtype='containers'
  AND regexp_like("$PATH",'application_1561661818238_0002')
  AND regexp_like(data, 'ERROR|WARN')
```

**Example – Querying the hadoop-Mapreduce partition for job job_1561661818238_0004 and failed reduces**

```sql
SELECT data,
```
Example – Querying Hive logs in the node partition for query ID 056e0609-33e1-4611-956c-7a31b42d2663

```
SELECT data, "$PATH"
FROM "default"."mypartitionedemrlogs"
WHERE logtype='node'
    AND regexp_like(data,'056e0609-33e1-4611-956c-7a31b42d2663') limit 100;
```

Example – Querying resourcemanager logs in the node partition for application 1567660019320_0001_01_000001

```
SELECT data, "$PATH"
FROM "default"."mypartitionedemrlogs"
WHERE logtype='node'
    AND regexp_like(data,'resourcemanager')
    AND regexp_like(data,'1567660019320_0001_01_000001') limit 100
```

**Querying AWS Global Accelerator flow logs**

You can use AWS Global Accelerator to create accelerators that direct network traffic to optimal endpoints over the AWS global network. For more information about Global Accelerator, see [What is AWS Global Accelerator](#).

Global Accelerator flow logs enable you to capture information about the IP address traffic going to and from network interfaces in your accelerators. Flow log data is published to Amazon S3, where you can retrieve and view your data. For more information, see [Flow logs in AWS Global Accelerator](#).

You can use Athena to query your Global Accelerator flow logs by creating a table that specifies their location in Amazon S3.

**To create the table for Global Accelerator flow logs**

1. Copy and paste the following DDL statement into the Athena console. This query specifies *ROW FORMAT DELIMITED* and omits specifying a *SerDe* (p. 302), which means that the query uses the *LazySimpleSerDe* (p. 324). In this query, fields are terminated by a space.

```
CREATE EXTERNAL TABLE IF NOT EXISTS aga_flow_logs (  
    version string,
    account string,
    acceleratorid string,
    clientip string,
    clientport int,
    gip string,
    gipport int,
    endpointip string,
    endpointport int,
    protocol string,
    ipaddresstype string,
    numpackets bigint,
    numbytes int,
    starttime int,
   endtime int,
)
```
action string,
logstatus string,
agasourceip string,
agasourceport int,
endpointregion string,
agaregion string,
direction string
)
PARTITIONED BY (dt string)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ' ' 
LOCATION 's3://your_log_bucket/prefix/AWSLogs/account_id/globalaccelerator/region/' 
TBLPROPERTIES ("skip.header.line.count"="1");

2. Modify the LOCATION value to point to the Amazon S3 bucket that contains your log data.

's3://your_log_bucket/prefix/AWSLogs/account_id/globalaccelerator/region_code/

3. Run the query in the Athena console. After the query completes, Athena registers the aga_flow_logs table, making the data in it available for queries.

4. Create partitions to read the data, as in the following sample query. The query creates a single partition for a specified date. Replace the placeholders for date and location.

```sql
ALTER TABLE aga_flow_logs
ADD PARTITION (dt='YYYY-MM-dd') 
LOCATION 's3://your_log_bucket/prefix/AWSLogs/account_id/
globalaccelerator/region_code/YYYY/MM/dd';
```

---

**Example queries for AWS Global Accelerator flow logs**

**Example – List the requests that pass through a specific edge location**

The following example query lists requests that passed through the LHR edge location. Use the LIMIT operator to limit the number of logs to query at one time.

```sql
SELECT 
  clientip, 
  agaregion, 
  protocol, 
  action 
FROM 
  aga_flow_logs 
WHERE 
  agaregion LIKE 'LHR%' 
LIMIT 100;
```

**Example – List the endpoint IP addresses that receive the most HTTPS requests**

To see which endpoint IP addresses are receiving the highest number of HTTPS requests, use the following query. This query counts the number of packets received on HTTPS port 443, groups them by destination IP address, and returns the top 10 IP addresses.

```sql
SELECT 
  SUM(numpackets) AS packetcount, 
  endpointip 
FROM 
  aga_flow_logs 
WHERE 
  443
```
Querying Amazon GuardDuty findings

Amazon GuardDuty is a security monitoring service for helping to identify unexpected and potentially unauthorized or malicious activity in your AWS environment. When it detects unexpected and potentially malicious activity, GuardDuty generates security findings that you can export to Amazon S3 for storage and analysis. After you export your findings to Amazon S3, you can use Athena to query them. This article shows how to create a table in Athena for your GuardDuty findings and query them.

For more information about Amazon GuardDuty, see the Amazon GuardDuty User Guide.

Prerequisites

- Enable the GuardDuty feature for exporting findings to Amazon S3. For steps, see Exporting findings in the Amazon GuardDuty User Guide.

Creating a table in Athena for GuardDuty findings

To query your GuardDuty findings from Athena, you must create a table for them.

To create a table in Athena for GuardDuty findings

2. Paste the following DDL statement into the Athena console. Modify the values in LOCATION 's3://findings-bucket-name/AWSLogs/account-id/GuardDuty/' to point to your GuardDuty findings in Amazon S3.

```
CREATE EXTERNAL TABLE `gd_logs` (  
    `schemaversion` string,  
    `accountid` string,  
    `region` string,  
    `partition` string,  
    `id` string,  
    `arn` string,  
    `type` string,  
    `resource` string,  
    `service` string,  
    `severity` string,  
    `createdat` string,  
    `updatedat` string,  
    `title` string,  
    `description` string) 
ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe' 
LOCATION 's3://findings-bucket-name/AWSLogs/account-id/GuardDuty/' 
TBLPROPERTIES ('has_encrypted_data'='true')
```

Note

The SerDe expects each JSON document to be on a single line of text with no line termination characters separating the fields in the record. If the JSON text is in pretty print format, you may receive an error message like HIVE_CURSOR_ERROR: Row is not a valid JSON Object or HIVE_CURSOR_ERROR: JsonParseException: Unexpected end-of-input:
expected close marker for OBJECT when you attempt to query the table after you create it. For more information, see JSON Data Files in the OpenX SerDe documentation on GitHub.

3. Run the query in the Athena console to register the gd_logs table. When the query completes, the findings are ready for you to query from Athena.

### Example queries

The following examples show how to query GuardDuty findings from Athena.

**Example – DNS data exfiltration**

The following query returns information about Amazon EC2 instances that might be exfiltrating data through DNS queries.

```sql
SELECT title,
       severity,
       type,
       id AS FindingID,
       accountid,
       region,
       createdat,
       updatedat,
       json_extract_scalar(service, '$.count') AS Count,
       json_extract_scalar(resource, '$.instancedetails.instanceid') AS InstanceID,
       json_extract_scalar(service, '$.action.actiontype') AS DNS_ActionType,
       json_extract_scalar(service, '$.action.dnsrequestaction.domain') AS DomainName,
       json_extract_scalar(service, '$.action.dnsrequestaction.protocol') AS protocol,
       json_extract_scalar(service, '$.action.dnsrequestaction.blocked') AS blocked
FROM gd_logs
WHERE type = 'Trojan:EC2/DNSDataExfiltration'
ORDER BY severity DESC
```

**Example – Unauthorized IAM user access**

The following query returns all UnauthorizedAccess:IAMUser finding types for an IAM Principal from all regions.

```sql
SELECT title,
       severity,
       type,
       id,
       accountid,
       region,
       createdat,
       updatedat,
       json_extract_scalar(service, '$.count') AS Count,
       json_extract_scalar(resource, '$.accesskeydetails.username') AS IAMPrincipal,
       json_extract_scalar(service,'$.action.awsapicallaction.api') AS APIActionCalled
FROM gd_logs
WHERE type LIKE '%UnauthorizedAccess:IAMUser%'
ORDER BY severity desc;
```

### Tips for querying GuardDuty findings

When you create your query, keep the following points in mind.

- To extract data from nested JSON fields, use the Presto `json_extract` or `json_extract_scalar` functions. For more information, see Extracting data from JSON (p. 427).
• Make sure that all characters in the JSON fields are in lower case.
• For information about downloading query results, see Downloading query results files using the Athena console (p. 347).

**Querying AWS Network Firewall logs**

AWS Network Firewall is a managed service that you can use to deploy essential network protections for your Amazon Virtual Private Cloud instances. AWS Network Firewall works together with AWS Firewall Manager so you can build policies based on AWS Network Firewall rules and then centrally apply those policies across your VPCs and accounts. For more information about AWS Network Firewall, see AWS Network Firewall.

You can configure AWS Network Firewall logging for traffic that you forward to your firewall’s stateful rules engine. Logging gives you detailed information about network traffic, including the time that the stateful engine received a packet, detailed information about the packet, and any stateful rule action taken against the packet. The logs are published to the log destination that you’ve configured, where you can retrieve and view them. For more information, see Logging network traffic from AWS Network Firewall in the AWS Network Firewall Developer Guide.

**To create the table for Network Firewall logs**

1. Copy and paste the following DDL statement into the Athena Query Editor. You may need to update the statement to include the columns for the latest version of the logs. For more information, see Contents of a firewall log in the AWS Network Firewall Developer Guide.

   **Note**
   Because `END` is a reserved word (p. 256) in Athena, the statement uses `finish:string` instead of `end:string`.

   ```sql
   CREATE EXTERNAL TABLE anf_logs(
     firewall_name string,
     availability_zone string,
     event_timestamp bigint,
     event struct<
       timestamp:string,
       flow_id:bigint,
       event_type:string,
       src_ip:string,
       src_port:int,
       dest_ip:string,
       dest_port:int,
       proto:string,
       app_proto:string,
       netflow:struct<
         pkts:int,
         bytes:bigint,
         start:string,
         finish:string,
         age:int,
         min_ttl:int,
         max_ttl:int
       >
     >
   )
   ROW FORMAT SERDE
   'org.openx.data.jsonserde.JsonSerDe'
   STORED AS INPUTFORMAT
   'org.apache.hadoop.mapred.TextInputFormat'
   OUTPUTFORMAT
   'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
   ```
2. Modify the LOCATION Amazon S3 bucket to specify the destination of your Network Firewall logs.
3. Run the query in the Athena console. After the query completes, Athena registers the `anf_logs` table, making the data in it ready for queries.

**Network Firewall example query**

The following query shows the count of requests evaluated by AWS Network Firewall that have unique source and destination IP pairs.

```sql
SELECT COUNT(*) AS count,
       event.src_ip,
       event.src_port,
       event.dest_ip,
       event.dest_port
FROM anf_logs
GROUP BY event.src_ip,
         event.src_port,
         event.dest_ip,
         event.dest_port
ORDER BY COUNT DESC
LIMIT 100
```

**Querying Network Load Balancer logs**

Use Athena to analyze and process logs from Network Load Balancer. These logs receive detailed information about the Transport Layer Security (TLS) requests sent to the Network Load Balancer. You can use these access logs to analyze traffic patterns and troubleshoot issues.

Before you analyze the Network Load Balancer access logs, enable and configure them for saving in the destination Amazon S3 bucket. For more information, see [Access logs for your Network Load Balancer](#).

- Create the table for Network Load Balancer logs (p. 470)
- Network Load Balancer example queries (p. 471)

**To create the table for Network Load Balancer logs**

1. Copy and paste the following DDL statement into the Athena console. Check the syntax of the Network Load Balancer log records. You may need to update the following query to include the columns and the Regex syntax for latest version of the record.

```sql
CREATE EXTERNAL TABLE IF NOT EXISTS nlb_tls_logs
(
  type string,
  version string,
  time string,
  elb string,
  listener_id string,
  client_ip string,
  client_port int,
  target_ip string,
  target_port int,
  tcp_connection_time_ms double,
  tls_handshake_time_ms double,
  received_bytes bigint,
  sent_bytes bigint,
) partitioned by (date string)
STORED AS parquet
LOCATION 's3://bucket_name/AWSLogs/
```
incoming_tls_alert int,
cert_arn string,
certificate_serial string,
tls_cipher_suite string,
tls_protocol_version string,
tls_named_group string,
domain_name string,
alpn_fe_protocol string,
alpn_be_protocol string,
alpn_client_preference_list string
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES (  
'serialization.format' = '1',
'input.regex' =  
'([\^ ]*) ([\^ ]*) ([\^ ]*) ([\^ ]*) ([\^ ]*) ([\^ ]*) ([0-9]*) ([0-9]*)  
([-.0-9]*) ([-.0-9]*) ([-.0-9]*) ([-.0-9]*) ([-.0-9]*) ([-.0-9]*) ([-.0-9]*) ([-.0-9]*)  
([-.0-9]*) ([-.0-9]*) ([-.0-9]*) ([-.0-9]*) ([-.0-9]*) ([-.0-9]*) ([-.0-9]*)$'
LOCATION 's3://your_log_bucket/prefix/AWSLogs/AWS_account_ID/  
elasticloadbalancing/region';

2. Modify the LOCATION Amazon S3 bucket to specify the destination of your Network Load Balancer logs.

3. Run the query in the Athena console. After the query completes, Athena registers the nlb_tls_logs table, making the data in it ready for queries.

Network Load Balancer example queries

To see how many times a certificate is used, use a query similar to this example:

```sql
SELECT count(*) AS ct, cert_arn
FROM "nlb_tls_logs"
GROUP BY cert_arn;
```

The following query shows how many users are using a TLS version earlier than 1.3:

```sql
SELECT tls_protocol_version,
       COUNT(tls_protocol_version) AS num_connections, client_ip
FROM "nlb_tls_logs"
WHERE tls_protocol_version < 'tlsv13'
GROUP BY tls_protocol_version, client_ip;
```

Use the following query to identify connections that take a long TLS handshake time:

```sql
SELECT *
FROM "nlb_tls_logs"
ORDER BY tls_handshake_time_ms DESC
LIMIT 10;
```

Use the following query to identify and count which TLS protocol versions and cipher suites have been negotiated in the past 30 days.

```sql
SELECT tls_cipher_suite,
       tls_protocol_version,
       COUNT(*) AS ct
FROM "nlb_tls_logs"
```
WHERE FROM_ISO8601_TIMESTAMP(time) > CURRENT_TIMESTAMP - INTERVAL '30' DAY
    AND NOT tls_protocol_version = '-'
GROUP BY tls_cipher_suite, tls_protocol_version
ORDER BY ct DESC;

**Querying Amazon Route 53 resolver query logs**

You can create Athena tables for your Amazon Route 53 Resolver query logs and query them from Athena.

Route 53 Resolver query logging is for logging of DNS queries made by resources within a VPC, on-premises resources that use inbound resolver endpoints, queries that use an outbound Resolver endpoint for recursive DNS resolution, and queries that use Route 53 Resolver DNS firewall rules to block, allow, or monitor a domain list. For more information about Resolver query logging, see [Resolver query logging](https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/resolver-logging.html) in the *Amazon Route 53 Developer Guide*. For information about each of the fields in the logs, see [Values that appear in resolver query logs](https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/resolver-logging-fields.html) in the *Amazon Route 53 Developer Guide*.

**Creating the table for resolver query logs**

You can use the Query Editor in the Athena console to create and query a table for your Route 53 Resolver query logs.

**To create and query an Athena table for Route 53 resolver query logs**

2. In the Athena Query Editor, enter the following `CREATE TABLE` statement. Replace the `LOCATION` clause values with those corresponding to the location of your Resolver logs in Amazon S3.

```sql
CREATE EXTERNAL TABLE r53_rlogs (  
  version string,
  account_id string,
  region string,
  vpc_id string,
  query_timestamp string,
  query_name string,
  query_type string,
  query_class string,
  rcode string,
  answers array<
    struct<
      Rdata: string,
      Type: string,
      Class: string
    >,  
  srcaddr string,
  srcport int,
  transport string,
  srcids struct<
    instance: string,
    resolver_endpoint: string
  >

)  
ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'
LOCATION 's3://your_log_bucket/AWSLogs/{account_id}/vpcdnsquerylogs/{vpc_id}/YYYY/MM/dd/';
```

Because Resolver query log data is in JSON format, the `CREATE TABLE` statement uses a [JSON SerDe library](https://docs.aws.amazon.com/AmazonS3/latest/userguide/json-serde.html) (p. 319) to analyze the data.
Note

The SerDe expects each JSON document to be on a single line of text with no line termination characters separating the fields in the record. If the JSON text is in pretty print format, you may receive an error message like HIVE_CURSOR_ERROR: Row is not a valid JSON Object or HIVE_CURSOR_ERROR: JsonParseException: Unexpected end-of-input: expected close marker for OBJECT when you attempt to query the table after you create it. For more information, see JSON Data Files in the OpenX SerDe documentation on GitHub.

3. Choose Run query. The statement creates an Athena table named r53_rlogs whose columns represent each of the fields in your Resolver log data.

4. In the Athena console Query Editor, run the following query to verify that your table has been created.

```
SELECT * FROM "r53_rlogs" LIMIT 10
```

Example queries

The following examples show some queries that you can perform from Athena on your Resolver query logs.

Example 1 - query logs in descending query_timestamp order

The following query displays log results in descending query_timestamp order.

```
SELECT * FROM "r53_rlogs"
ORDER BY query_timestamp DESC
```

Example 2 - query logs within specified start and end times

The following query queries logs between midnight and 8am on September 24, 2020. Substitute the start and end times according to your own requirements.

```
SELECT query_timestamp, srcids.instance, srcaddr, srcport, query_name, rcode
FROM "r53_rlogs"
WHERE (parse_datetime(query_timestamp,'yyyy-MM-dd''T''HH:mm:ss''Z')
BETWEEN parse_datetime('2020-09-24-00:00:00','yyyy-MM-dd-HH:mm:ss')
AND parse_datetime('2020-09-24-08:00:00','yyyy-MM-dd-HH:mm:ss'))
ORDER BY query_timestamp DESC
```

Example 3 - query logs based on a specified DNS query name pattern

The following query selects records whose query name includes the string "example.com".

```
SELECT query_timestamp, srcids.instance, srcaddr, srcport, query_name, rcode, answers
FROM "r53_rlogs"
WHERE query_name LIKE '%example.com%'
ORDER BY query_timestamp DESC
```

Example 4 - query log requests with no answer

The following query selects log entries in which the request received no answer.

```
SELECT query_timestamp, srcids.instance, srcaddr, srcport, query_name, rcode, answers
FROM "r53_rlogs"
WHERE cardinality(answers) = 0
```
Example 5 - query logs with a specific answer

The following query shows logs in which the answer.Rdata value has the specified IP address.

```sql
SELECT query_timestamp, srcids.instance, srcaddr, srcport, query_name, rcode, answer.Rdata
FROM "r53_rlogs"
CROSS JOIN UNNEST(r53_rlogs.answers) as st(answer)
WHERE answer.Rdata='203.0.113.16';
```

Querying Amazon SES event logs

You can use Amazon Athena to query Amazon Simple Email Service (Amazon SES) event logs.

Amazon SES is an email platform that provides a convenient and cost-effective way to send and receive email using your own email addresses and domains. You can monitor your Amazon SES sending activity at a granular level using events, metrics, and statistics.

Based on the characteristics that you define, you can publish Amazon SES events to Amazon CloudWatch, Amazon Kinesis Data Firehose, or Amazon Simple Notification Service. After the information is stored in Amazon S3, you can query it from Amazon Athena.

For more information about how to analyze Amazon SES email events using Kinesis Data Firehose, Amazon Athena, and Amazon QuickSight, see Analyzing Amazon SES event data with AWS Analytics Services in the AWS Messaging and Targeting Blog.

Querying Amazon VPC flow logs

Amazon Virtual Private Cloud flow logs capture information about the IP traffic going to and from network interfaces in a VPC. Use the logs to investigate network traffic patterns and identify threats and risks across your VPC network.

To query your Amazon VPC flow logs, you have two options:

- **Amazon VPC Console** – Use the Athena integration feature in the Amazon VPC Console to generate an AWS CloudFormation template that creates an Athena database, workgroup, and flow logs table with partitioning for you. The template also creates a set of predefined flow log queries that you can use to obtain insights about the traffic flowing through your VPC.

  For information about this approach, see Query flow logs using Amazon Athena in the Amazon VPC User Guide.

- **Amazon Athena console** – Create your tables and queries directly in the Athena console. For more information, continue reading this page.

Creating and querying tables for VPC flow logs

Before you begin querying the logs in Athena, enable VPC flow logs, and configure them to be saved to your Amazon S3 bucket. After you create the logs, let them run for a few minutes to collect some data. The logs are created in a GZIP compression format that Athena lets you query directly.

When you create a VPC flow log, you can use the default format or specify a custom format. Use a custom format when you want to specify the fields to return in the flow log and the order in which the fields appear. For more information, see Flow log records in the Amazon VPC User Guide.

Common considerations

When you create tables in Athena for Amazon VPC flow logs, remember the following points:
• By default, in Athena, Parquet will access columns by name. For more information, see Handling schema updates (p. 376).

• Use the names in the flow log records for the column names in Athena. The names of the columns in the Athena schema should exactly match the field names in the Amazon VPC flow logs, with the following differences:
  • Replace the hyphens in the Amazon VPC log field names with underscores in the Athena column names. In Athena, the only acceptable characters for database names, table names, and column names are lowercase letters, numbers, and the underscore character. For more information, see Database, table, and column names (p. 26).
  • Escape the flow log record names that are reserved keywords (p. 256) in Athena by enclosing them with backticks.
  • VPC flow logs are AWS account specific. When you publish your log files to Amazon S3, the path that Amazon VPC creates in Amazon S3 includes the ID of the AWS account that was used to create the flow log. For more information, see Publish flow logs to Amazon S3 in the Amazon VPC User Guide.

CREATE TABLE statement for Amazon VPC flow logs

The following procedure creates an Amazon VPC table for Amazon VPC flow logs. If you create a flow log with a custom format, you must create a table with fields that match the fields that you specified when you created the flow log in the same order that you specified them.

To create an Athena table for Amazon VPC flow logs

1. Enter a DDL statement like the following into the Athena console query editor, following the guidelines in the Common considerations (p. 474) section. The sample statement creates a table that has the columns for Amazon VPC flow logs versions 2 through 5 as documented in Flow log records. If you use a different set of columns or order of columns, modify the statement accordingly.

```sql
CREATE EXTERNAL TABLE IF NOT EXISTS `vpc_flow_logs` (
  'version' int,
  'account_id' string,
  'interface_id' string,
  'srcaddr' string,
  'dstaddr' string,
  'srcport' int,
  'dstport' int,
  'protocol' bigint,
  'packets' bigint,
  'bytes' bigint,
  'start' bigint,
  'end' bigint,
  'action' string,
  'log_status' string,
  'vpc_id' string,
  'subnet_id' string,
  'instance_id' string,
  'tcp_flags' int,
  'type' string,
  'pkt_srcaddr' string,
  'pkt_dstaddr' string,
  'region' string,
  'az_id' string,
  'sublocation_type' string,
  'sublocation_id' string,
  'pkt_src_aws_service' string,
  'pkt_dst_aws_service' string,
  'flow_direction' string,
  'traffic_path' int
)"
```
PARTITIONED BY (`date` date)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ' '
LOCATION 's3://DOC-EXAMPLE-BUCKET/prefix/AWSLogs/{account_id}/vpcflowlogs/{region_code}/'
TBLPROPERTIES ("skip.header.line.count"="1");

Note the following points:

• The query specifies ROW FORMAT DELIMITED and omits specifying a SerDe. This means that the query uses the LazySimpleSerDe for CSV, TSV, and custom-delimited files (p. 324). In this query, fields are terminated by a space.

• The PARTITIONED BY clause uses the date type. This makes it possible to use mathematical operators in queries to select what's older or newer than a certain date.

  Note
  Because date is a reserved keyword in DDL statements, it is escaped by backtick characters. For more information, see Reserved keywords (p. 256).

• For a VPC flow log with a custom format, modify the fields to match the fields that you specified when you created the flow log.

2. Modify the LOCATION 's3://DOC-EXAMPLE-BUCKET/prefix/AWSLogs/{account_id}/vpcflowlogs/{region_code}/' to point to the Amazon S3 bucket that contains your log data.

3. Run the query in Athena console. After the query completes, Athena registers the vpc_flow_logs table, making the data in it ready for you to issue queries.

4. Create partitions to be able to read the data, as in the following sample query. This query creates a single partition for a specified date. Replace the placeholders for date and location as needed.

  Note
  This query creates a single partition only, for a date that you specify. To automate the process, use a script that runs this query and creates partitions this way for the year/month/day, or use a CREATE TABLE statement that specifies partition projection (p. 478).

```
ALTER TABLE vpc_flow_logs
ADD PARTITION (`date`='YYYY-MM-dd')
LOCATION 's3://DOC-EXAMPLE-BUCKET/prefix/AWSLogs/{account_id}/vpcflowlogs/{region_code}/YYYY/MM/dd';
```

Example queries for the vpc_flow_logs table

Use the query editor in the Athena console to run SQL statements on the table that you create. You can save the queries, view previous queries, or download query results in CSV format. In the following examples, replace vpc_flow_logs with the name of your table. Modify the column values and other variables according to your requirements.

The following example query lists a maximum of 100 flow logs for the date specified.

```
SELECT *
FROM vpc_flow_logs
WHERE date = DATE('2020-05-04')
LIMIT 100;
```

The following query lists all of the rejected TCP connections and uses the newly created date partition column, date, to extract from it the day of the week for which these events occurred.

```
SELECT day_of_week(date) AS
```

476
To see which one of your servers is receiving the highest number of HTTPS requests, use the following query. It counts the number of packets received on HTTPS port 443, groups them by destination IP address, and returns the top 10 from the last week.

```
SELECT SUM(packets) AS packetcount, dstaddr
FROM vpc_flow_logs
WHERE dstport = 443 AND date > current_date - interval '7' day
GROUP BY dstaddr
ORDER BY packetcount DESC
LIMIT 10;
```

Creating tables for flow logs in Apache Parquet format

The following procedure creates an Amazon VPC table for Amazon VPC flow logs in Apache Parquet format.

To create an Athena table for Amazon VPC flow logs in Parquet format

1. Enter a DDL statement like the following into the Athena console query editor, following the guidelines in the Common considerations (p. 474) section. The sample statement creates a table that has the columns for Amazon VPC flow logs versions 2 through 5 as documented in Flow log records in Parquet format, Hive partitioned hourly. If you do not have hourly partitions, remove hour from the PARTITIONED BY clause.

```
CREATE EXTERNAL TABLE IF NOT EXISTS vpc_flow_logs_parquet (  
    `version` int,  
    `account_id` string,  
    `interface_id` string,  
    `srcaddr` string,  
    `dstaddr` string,  
    `srcport` int,  
    `dstport` int,  
    `protocol` bigint,  
    `packets` bigint,  
    `bytes` bigint,  
    `start` bigint,  
    `end` bigint,  
    `action` string,  
    `log_status` string,  
    `vpc_id` string,  
    `subnet_id` string,  
    `instance_id` string,  
    `tcp_flags` int,  
    `type` string,  
    `pkt_srcaddr` string,  
    `pkt_dstaddr` string,  
    `region` string,  
    `az_id` string,  
    `sublocation_type` string,
```
'sublocation_id' string,
'pkt_src_aws_service' string,
'pkt_dst_aws_service' string,
'flow_direction' string,
'traffic_path' int
)
PARTITIONED BY (
    'aws-account-id' string,
    'aws-service' string,
    'aws-region' string,
    'year' string,
    'month' string,
    'day' string,
    'hour' string
)
ROW FORMAT SERDE
    'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
STORED AS INPUTFORMAT
    'org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat'
OUTPUTFORMAT
    'org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat'
LOCATION
    's3://DOC-EXAMPLE-BUCKET/parquet/AWSLogs/aws-account-id=account_id/aws-service=vpcflowlogs/aws-region=region_code/
TBLPROPERTIES (    'EXTERNAL'='true',
    'skip.header.line.count'='1'
)

2. Modify the sample LOCATION 's3://DOC-EXAMPLE-BUCKET/parquet/AWSLogs/aws-account-id=account_id/aws-service=vpcflowlogs/aws-region=region_code/' to point to the Amazon S3 path that contains your log data.

3. Run the query in Athena console.

4. If your data is in Hive-compatible format, run the following command in the Athena console to update and load the Hive partitions in the metastore. After the query completes, you can query the data in the vpc_flow_logs_parquet table.

    MSCK REPAIR TABLE vpc_flow_logs_parquet

If you are not using Hive compatible data, run ALTER TABLE ADD PARTITION (p. 733) to load the partitions.

For more information about using Athena to query Amazon VPC flow logs in Parquet format, see the post Optimize performance and reduce costs for network analytics with VPC Flow Logs in Apache Parquet format in the AWS Big Data Blog.

Creating and querying a table for Amazon VPC flow logs using partition projection

Use a CREATE TABLE statement like the following to create a table, partition the table, and populate the partitions automatically by using partition projection (p. 265). Replace the table name test_table_vpclogs in the example with the name of your table. Edit the LOCATION clause to specify the Amazon S3 bucket that contains your Amazon VPC log data.

The following CREATE TABLE statement is for VPC flow logs delivered in non-Hive style partitioning format.

    CREATE EXTERNAL TABLE IF NOT EXISTS test_table_vpclogs (    'version' int,
Example queries for test_table_vpclogs

The following example queries query the test_table_vpclogs created by the preceding CREATE TABLE statement. Replace test_table_vpclogs in the queries with the name of your own table. Modify the column values and other variables according to your requirements.

To return the first 100 access log entries in chronological order for a specified period of time, run a query like the following.

```sql
SELECT *
FROM test_table_vpclogs
WHERE day >= '2021/02/01' AND day < '2021/02/28'
ORDER BY day ASC
LIMIT 100
```

To view which server receives the top ten number of HTTP packets for a specified period of time, run a query like the following. The query counts the number of packets received on HTTPS port 443, groups them by destination IP address, and returns the top 10 entries from the previous week.
To return the packet count for a destination address within a specified date range, run a query like the following:

```sql
SELECT SUM(packets) AS packetcount,
       dstaddr
FROM test_table_vpclogs
WHERE dstport = 443
  AND day >= '2021/03/01'
  AND day < '2021/03/31'
GROUP BY dstaddr
ORDER BY packetcount DESC
LIMIT 10
```

To return the logs that were created during a specified period of time, run a query like the following:

```sql
SELECT interface_id,
       srcaddr,
       action,
       protocol,
       to_iso8601(from_unixtime(start)) AS start_time,
       to_iso8601(from_unixtime("end")) AS end_time
FROM test_table_vpclogs
WHERE DAY >= '2021/04/01'
  AND DAY < '2021/04/30'
```

To return the access logs for a source IP address between specified time periods, run a query like the following:

```sql
SELECT *
FROM test_table_vpclogs
WHERE srcaddr = '10.117.1.22'
  AND day >= '2021/02/01'
  AND day < '2021/02/28'
```

To list rejected TCP connections, run a query like the following:

```sql
SELECT day,
       interface_id,
       srcaddr,
       action,
       protocol
FROM test_table_vpclogs
WHERE action = 'REJECT' AND protocol = 6 AND day >= '2021/02/01' AND day < '2021/02/28'
LIMIT 10
```

To return the access logs for the IP address range that starts with 10.117, run a query like the following:

```sql
SELECT *
FROM test_table_vpclogs
WHERE split_part(srcaddr,'.', 1)='10'
  AND split_part(srcaddr,'.', 2) = '117'
```

To return the access logs for a destination IP address between a certain time range, run a query like the following:

```sql
SELECT *
FROM test_table_vpclogs
WHERE dstaddr = '10.0.1.14'
  AND day >= '2021/01/01'
  AND day < '2021/01/31'
```
Additional resources

For more information about using Athena to analyze VPC flow logs, see the AWS Big Data blog post Analyzing VPC flow logs using Amazon Athena and Amazon QuickSight.

Querying AWS WAF logs

AWS WAF logs include information about the traffic that is analyzed by your web ACL, such as the time that AWS WAF received the request from your AWS resource, detailed information about the request, and the action for the rule that each request matched.

You can enable access logging for AWS WAF logs and save them to Amazon S3. Make a note of the Amazon S3 bucket to which you save these logs, and you can create an Athena table for them and query them in Athena.

For more information about enabling AWS WAF logs and about the log record structure, see Logging and monitoring web ACL traffic in the AWS WAF Developer Guide.

For information about individual AWS WAF log fields, see Log fields in the AWS WAF Developer Guide.

For an example of how to aggregate AWS WAF logs into a central data lake repository and query them with Athena, see the AWS Big Data Blog post Analyzing AWS WAF logs with OpenSearch Service, Amazon Athena, and Amazon QuickSight.

This topic provides two example CREATE TABLE statements: one that uses partitioning and one that does not.

Note

The CREATE TABLE statements in this topic can be used for both v1 and v2 AWS WAF logs.

In v1, the webaclid field contains an ID. In v2, the webaclid field contains a full ARN. The CREATE TABLE statements here treat this content agnostically by using the string data type.

Topics

- Creating the table for AWS WAF logs in Athena using partition projection (p. 481)
- Creating a table for AWS WAF logs without partitioning (p. 484)
- Example queries for AWS WAF logs (p. 486)

Creating the table for AWS WAF logs in Athena using partition projection

Because AWS WAF logs have a known structure whose partition scheme you can specify in advance, you can reduce query runtime and automate partition management by using the Athena partition projection (p. 265) feature. Partition projection automatically adds new partitions as new data is added. This removes the need for you to manually add partitions by using ALTER TABLE ADD PARTITION.

The following example CREATE TABLE statement automatically uses partition projection on AWS WAF logs from a specified date until the present for four different AWS regions. The PARTITION BY clause in this example partitions by region and by date, but you can modify this according to your requirements. Modify the fields as necessary to match your log output. In the LOCATION and storage.location.template clauses, replace the bucket and accountID placeholders with values that identify the Amazon S3 bucket location of your AWS WAF logs. For projection.day.range, replace 2021/01/01 with the starting date that you want to use. After you run the query successfully, you can query the table. You do not have to run ALTER TABLE ADD PARTITION to load the partitions.

```sql
CREATE EXTERNAL TABLE `waf_logs`(
    `timestamp` bigint,
```
'formatversion' int,
'webaclid' string,
'terminatingruleid' string,
'terminatingruletype' string,
'action' string,
'terminatingrulematchdetails' array <
    struct <
        conditiontype: string,
        sensitivitylevel: string,
        location: string,
        matcheddata: array < string >
    >,
>,
'hostname' string,
'hitcount' int,
'rate' float,
'time' int,
'terminatingruleid' string,
'action' string,
'terminatingrulematchdetails' array <
    struct <
        conditiontype: string,
        sensitivitylevel: string,
        location: string,
        matcheddata: array < string >
    >,
>,
'ratebasedruleid' string,
'limitkey' string,
'maxrateallowed' int
>,
'nonterminatingruleid' string,
'action' string,
'rulematchdetails' array <
    struct <
        conditiontype: string,
        sensitivitylevel: string,
        location: string,
        matcheddata: array < string >
    >,
>,
'excludedrules' string
>,
'ratebasedruleid' string,
'limitkey' string,
'maxrateallowed' int
>,
'nonterminatingruleid' string,
'action' string,
'rulematchdetails' array <
    struct <
        conditiontype: string,
        sensitivitylevel: string,
        location: string,
        matcheddata: array < string >
    >,
>,
'ratebasedruleid' string,
'limitkey' string,
'maxrateallowed' int
>,
PARTITIONED BY (region string, date string)
ROW FORMAT SERDE
'org.openx.data.jsonserde.JsonSerDe'
STORED AS INPUTFORMAT
'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION
's3://bucket/AWSLogs/accountID/WAFLogs/region/webACL/'
TBLPROPERTIES(
'projection.enabled' = 'true',
'projection.region.type' = 'enum',
'projection.region.values' = 'us-east-1,us-west-2,eu-central-1,eu-west-1',
'projection.date.type' = 'date',
'projection.date.range' = '2021/01/01,NOW',

action: string,
rulematchdetails: array <
  struct <
    conditiontype: string,
    sensitivitylevel: string,
    location: string,
    matcheddata: array <
      string >
  >,
  captcharesponse: struct <
    responsecode: string,
    solvetimestamp: string
  >
>,
'requestheadersinserted' array <
  struct <
    name: string,
    value: string
  >
>,
'reponsecodesent' string,
'httprequest' struct <
  clientip: string,
  country: string,
  headers: array <
    struct <
      name: string,
      value: string
    >
  >,
  uri: string,
  args: string,
  httpversion: string,
  httpmethod: string,
  requestid: string
>,
'labels' array <
  struct <
    name: string
  >
>,
'captcharesponse' struct <
  responsecode: string,
  solvetimestamp: string,
  failureReason: string
>
'projection.date.format' = 'yyyy/MM/dd',
'projection.date.interval' = '1',
'projection.date.interval.unit' = 'DAYS',
'storage.location.template' = 's3://DOC-EXAMPLE-BUCKET/AWSLogs/accountID/WAFLogs/
${region}/webACL/${date}/')

Note
The format of the path in the LOCATION clause in the example is standard but can vary based on the AWS WAF configuration that you have implemented. For example, the following example AWS WAF logs path is for a CloudFront distribution:

s3://DOC-EXAMPLE-BUCKET/AWSLogs/12345678910/WAFLogs/cloudfront/cloudfronyt/2022/08/08/17/55/

If you experience issues while creating or querying your AWS WAF logs table, confirm the location of your log data or contact AWS Support.

For more information about partition projection, see Partition projection with Amazon Athena (p. 265).

Creating a table for AWS WAF logs without partitioning

This section describes how to create a table for AWS WAF logs without partitioning or partition projection.

To create the AWS WAF table

1. Copy and paste the following DDL statement into the Athena console. Modify the fields as necessary to match your log output. Modify the LOCATION for the Amazon S3 bucket to correspond to the one that stores your logs.

This query uses the OpenX JSON SerDe (p. 321).

Note
The SerDe expects each JSON document to be on a single line of text with no line termination characters separating the fields in the record. If the JSON text is in pretty print format, you may receive an error message like HIVE_CURSOR_ERROR: Row is not a valid JSON Object or HIVE_CURSOR_ERROR: JsonParseException: Unexpected end-of-input: expected close marker for OBJECT when you attempt to query the table after you create it. For more information, see JSON Data Files in the OpenX SerDe documentation on GitHub.

CREATE EXTERNAL TABLE `waf_logs`(
    `timestamp` bigint,
    `formatversion` int,
    `webaclid` string,
    `terminatingruleid` string,
    `terminatingruletype` string,
    `action` string,
    `terminatingrulematchdetails` array <
        struct <
            conditiontype: string,
            sensitivitylevel: string,
            location: string,
            matcheddata: array < string >
        >
    >,
    `httpsourcename` string,
    `httpsourceid` string,
    `rulegrouplist` array <
        struct <
            rulegroupid: string,
            ...
terminatingrule: struct {
    ruleid: string,
    action: string,
    rulematchdetails: array <
        struct {
            conditiontype: string,
            sensitivitylevel: string,
            location: string,
            matcheddata: array < string >
        }
    >
},
nonterminatingmatchingrules: array <
    struct {
        ruleid: string,
        action: string,
        rulematchdetails: array <
            struct {
                conditiontype: string,
                sensitivitylevel: string,
                location: string,
                matcheddata: array < string >
            }
        >
    }
>,
excludedrules: string

'ratebasedrulelist' array <
    struct {
        ratebasedruleid: string,
        limitkey: string,
        maxrateallowed: int
    }
>,
'nonterminatingmatchingrules' array <
    struct {
        ruleid: string,
        action: string,
        rulematchdetails: array <
            struct {
                conditiontype: string,
                sensitivitylevel: string,
                location: string,
                matcheddata: array < string >
            }
        >
    }
>,
captcharesponse: struct {
    responsecode: string,
    solvetimestamp: string
}
2. Run the CREATE EXTERNAL TABLE statement in the Athena console query editor. This registers the `waf_logs` table and makes the data in it available for queries from Athena.

Example queries for AWS WAF logs

The following example queries the partition projection table created previously in this document. Modify the table name, column values, and other variables in the examples according to your requirements. To improve the performance of your queries and reduce cost, add the partition column in the filter condition.

Count the number of referrers that contain a specified term

The following query counts the number of referrers that contain the term "amazon" for the specified date range.

```
WITH test_dataset AS
(SELECT header FROM waf_logs
  CROSS JOIN UNNEST(httprequest.headers) AS t(header) WHERE "date" >= '2021/03/01'
  AND "date" < '2021/03/31')
SELECT COUNT(*) referer_count
FROM test_dataset
WHERE LOWER(header.name)='referer' AND header.value LIKE '%amazon%'
```

Count all matched IP addresses in the last 10 days that have matched excluded rules
The following query counts the number of times in the last 10 days that the IP address matched the excluded rule in the rule group.

```sql
WITH test_dataset AS
(SELECT * FROM waf_logs
  CROSS JOIN UNNEST(rulegrouplist) AS t(allrulegroups))
SELECT COUNT(*) AS count,
  "httprequest"."clientip",
  "allrulegroups"."excludedrules",
  "allrulegroups"."ruleGroupId"
FROM test_dataset
WHERE allrulegroups.excludedrules IS NOT NULL AND from_unixtime(timestamp/1000) > now() - interval '10' day
GROUP BY "httprequest"."clientip", "allrulegroups"."ruleGroupId",
"allrulegroups"."excludedrules"
ORDER BY count DESC
```

### Working with date and time

**Return the timestamp field in human-readable ISO 8601 format**

The following query uses the `from_unixtime` and `to_iso8601` functions to return the timestamp field in human-readable ISO 8601 format (for example, 2019-12-13T23:40:12.000Z instead of 1576280412771). The query also returns the HTTP source name, source ID, and request.

```sql
SELECT to_iso8601(from_unixtime(timestamp / 1000)) as time_ISO_8601,
  httpsourcename,
  httpsourceid,
  httprequest
FROM waf_logs
LIMIT 10;
```

**Return records from the last 24 hours**

The following query uses a filter in the `WHERE` clause to return the HTTP source name, HTTP source ID, and HTTP request fields for records from the last 24 hours.

```sql
SELECT to_iso8601(from_unixtime(timestamp/1000)) AS time_ISO_8601,
  httpsourcename,
  httpsourceid,
  httprequest
FROM waf_logs
WHERE from_unixtime(timestamp/1000) > now() - interval '1' day
LIMIT 10;
```

**Return records for a specified date range and IP address**

The following query lists the records in a specified date range for a specified client IP address.

```sql
SELECT *
FROM waf_logs
WHERE httprequest.clientip='53.21.198.66' AND "date" >= '2021/03/01' AND "date" < '2021/03/31'
```

**For a specified date range, count the number of IP addresses in five minute intervals**

The following query counts, for a particular date range, the number of IP addresses in five minute intervals.

```sql
WITH test_dataset AS
```
Count the number of X-Forwarded-For IP in the last 10 days

The following query filters the request headers and counts the number of X-Forwarded-For IP in the last 10 days.

```sql
WITH test_dataset AS
(SELECT header
FROM waf_logs
CROSS JOIN UNNEST (httprequest.headers) AS t(header)
WHERE from_unixtime("timestamp"/1000) > now() - interval '10' DAY)
SELECT header.value AS ip,
count(*) AS COUNT
FROM test_dataset
WHERE header.name='X-Forwarded-For'
GROUP BY header.value
ORDER BY COUNT DESC
```

For more information about date and time functions, see [Date and time functions and operators](#) in the Trino documentation.

Working with blocked requests and addresses

Extract the top 100 IP addresses blocked by a specified rule type

The following query extracts and counts the top 100 IP addresses that have been blocked by the RATE_BASED terminating rule during the specified date range.

```sql
SELECT COUNT(httpRequest.clientIp) as count,
httpRequest.clientIp
FROM waf_logs
WHERE terminatingruletype='RATE_BASED' AND action='BLOCK' and "date" >= '2021/03/01'
AND "date" < '2021/03/31'
GROUP BY httpRequest.clientIp
ORDER BY count DESC
LIMIT 100
```

Count the number of times a request from a specified country has been blocked

The following query counts the number of times the request has arrived from an IP address that belongs to Ireland (IE) and has been blocked by the RATE_BASED terminating rule.

```sql
SELECT COUNT(httpRequest.country) as count,
httpRequest.country
FROM waf_logs
WHERE terminatingruletype='RATE_BASED' AND httpRequest.country='IE'
GROUP BY httpRequest.country
ORDER BY count
LIMIT 100;
```

Count the number of times a request has been blocked, grouping by specific attributes
The following query counts the number of times the request has been blocked, with results grouped by WebACL, RuleId, ClientIP, and HTTP Request URI.

```
SELECT COUNT(*) AS count, 
    webaclid, 
    terminatingruleid, 
    httprequest.clientip, 
    httprequest.uri 
FROM waf_logs 
WHERE action='BLOCK' 
GROUP BY webaclid, terminatingruleid, httprequest.clientip, httprequest.uri 
ORDER BY count DESC 
LIMIT 100;
```

**Count the number of times a specific terminating rule ID has been matched**

The following query counts the number of times a specific terminating rule ID has been matched (WHERE terminatingruleid='e9dd190d-7a43-4c06-bcea-409613d9506e'). The query then groups the results by WebACL, Action, ClientIP, and HTTP Request URI.

```
SELECT COUNT(*) AS count, 
    webaclid, 
    action, 
    httprequest.clientip, 
    httprequest.uri 
FROM waf_logs 
WHERE terminatingruleid='e9dd190d-7a43-4c06-bcea-409613d9506e' 
GROUP BY webaclid, action, httprequest.clientip, httprequest.uri 
ORDER BY count DESC 
LIMIT 100;
```

**Retrieve the top 100 IP addresses blocked during a specified date range**

The following query extracts the top 100 IP addresses that have been blocked for a specified date range. The query also lists the number of times the IP addresses have been blocked.

```
SELECT "httprequest"."clientip", "count"(*) "ipcount", "httprequest"."country" 
FROM waf_logs 
WHERE "action" = 'BLOCK' and "date" >= '2021/03/01' 
AND "date" < '2021/03/31' 
GROUP BY "httprequest"."clientip", "httprequest"."country" 
ORDER BY "ipcount" DESC limit 100
```

For information about querying Amazon S3 logs, see the following topics:

- How do I analyze my Amazon S3 server access logs using Athena? in the AWS Knowledge Center
- Querying Amazon S3 access logs for requests using Amazon Athena in the Amazon Simple Storage Service User Guide
- Using AWS CloudTrail to identify Amazon S3 requests in the Amazon Simple Storage Service User Guide

**Querying web server logs stored in Amazon S3**

You can use Athena to query Web server logs stored in Amazon S3. The topics in this section show you how to create tables in Athena to query Web server logs in a variety of formats.
Querying Apache logs stored in Amazon S3

You can use Amazon Athena to query Apache HTTP Server log files stored in your Amazon S3 account. This topic shows you how to create table schemas to query Apache Access log files in the common log format.

Fields in the common log format include the client IP address, client ID, user ID, request received timestamp, text of the client request, server status code, and size of the object returned to the client.

The following example data shows the Apache common log format.

```
198.51.100.7 - Li [10/Oct/2019:13:55:36 -0700] "GET /logo.gif HTTP/1.0" 200 232
198.51.100.22 - Mateo [27/Dec/2019:11:38:12 -0700] "GET /about.html HTTP/1.1" 200 1287
198.51.100.2 - Ana [15/Feb/2019:10:12:22 -0700] "GET /favicon.ico HTTP/1.1" 404 30
198.51.100.13 - Saanvi [14/Mar/2019:11:40:33 -0700] "GET /intro.html HTTP/1.1" 200 1608
198.51.100.11 - Xiulan [22/Apr/2019:10:51:34 -0700] "GET /group/index.html HTTP/1.1" 200 1344
```

Creating a table in Athena for Apache logs

Before you can query Apache logs stored in Amazon S3, you must create a table schema for Athena so that it can read the log data. To create an Athena table for Apache logs, you can use the Grok SerDe (p. 317). For more information about using the Grok SerDe, see Writing grok custom classifiers in the AWS Glue Developer Guide.

To create a table in Athena for Apache web server logs

2. Paste the following DDL statement into the Athena Query Editor. Modify the values in LOCATION 's3://bucket-name/apache-log-folder/' to point to your Apache logs in Amazon S3.

```
CREATE EXTERNAL TABLE apache_logs(
  client_ip string,
  client_id string,
  user_id string,
  request_received_time string,
  client_request string,
  server_status string,
  returned_obj_size string
)
ROW FORMAT SERDE
  'com.amazonaws.glue.serde.GrokSerDe'
WITH SERDEPROPERTIES (
  'input.format'='\%{IPV4:client_ip} \%{DATA:client_id} \%{USERNAME:user_id} \%{GREEDYDATA:request_received_time} \%{QUOTEDSTRING:client_request} \%{DATA:server_status} \%{DATA: returned_obj_size}'
)
STORED AS INPUTFORMAT
  'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
```
LOCATION 's3://bucket-name/apache-log-folder/';

3. Run the query in the Athena console to register the apache_logs table. When the query completes, the logs are ready for you to query from Athena.

**Example select queries for Apache logs**

**Example – Filtering for 404 errors**

The following example query selects the request received time, text of the client request, and server status code from the apache_logs table. The WHERE clause filters for HTTP status code 404 (page not found).

```sql
SELECT request_received_time, client_request, server_status
FROM apache_logs
WHERE server_status = '404'
```

The following image shows the results of the query in the Athena Query Editor.

![Query Results](image)

**Example – Filtering for successful requests**

The following example query selects the user ID, request received time, text of the client request, and server status code from the apache_logs table. The WHERE clause filters for HTTP status code 200 (successful).

```sql
SELECT user_id, request_received_time, client_request, server_status
FROM apache_logs
WHERE server_status = '200'
```

The following image shows the results of the query in the Athena Query Editor.

![Query Results](image)
Querying internet information server (IIS) logs stored in Amazon S3

You can use Amazon Athena to query Microsoft Internet Information Services (IIS) web server logs stored in your Amazon S3 account. While IIS uses a variety of log file formats, this topic shows you how to create table schemas to query W3C extended and IIS log file format logs from Athena.

Because the W3C extended and IIS log file formats use single character delimiters (spaces and commas, respectively) and do not have values enclosed in quotation marks, you can use the LazySimpleSerDe (p. 324) to create Athena tables for them.

W3C extended log file format

The W3C extended log file data format has space-separated fields. The fields that appear in W3C extended logs are determined by a web server administrator who chooses which log fields to include. The following example log data has the fields date, time, c-ip, s-ip, cs-method, cs-uri-stem, sc-status, sc-bytes, cs-bytes, time-taken, and cs-version.

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Client IP</th>
<th>Server IP</th>
<th>Request</th>
<th>Status</th>
<th>Bytes</th>
<th>Client Bytes</th>
<th>Server Bytes</th>
<th>Time Taken</th>
<th>Server Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>2020-01-19</td>
<td>22:48:39</td>
<td>203.0.113.5</td>
<td>198.51.100.2</td>
<td>GET /default.html</td>
<td>200</td>
<td>540</td>
<td>524</td>
<td>157</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2020-01-19</td>
<td>22:49:40</td>
<td>203.0.113.10</td>
<td>198.51.100.12</td>
<td>GET /index.html</td>
<td>200</td>
<td>420</td>
<td>324</td>
<td>164</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2020-01-19</td>
<td>22:51:44</td>
<td>203.0.113.12</td>
<td>198.51.100.4</td>
<td>GET /image.gif</td>
<td>200</td>
<td>330</td>
<td>324</td>
<td>358</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Creating a table in Athena for W3C extended logs

Before you can query your W3C extended logs, you must create a table schema so that Athena can read the log data.

To create a table in Athena for W3C extended logs

2. Paste a DDL statement like the following into the Athena console, noting the following points:
   a. Add or remove the columns in the example to correspond to the fields in the logs that you want to query.
   b. Column names in the W3C extended log file format contain hyphens (-). However, in accordance with Athena naming conventions (p. 254), the example CREATE TABLE statement replaces them with underscores (_).
   c. To specify the space delimiter, use FIELDS TERMINATED BY ' '.
   d. Modify the values in LOCATION 's3://bucket-name/w3c-log-folder/' to point to your W3C extended logs in Amazon S3.

```sql
CREATE EXTERNAL TABLE `iis_w3c_logs`(
    date_col string,
    time_col string,
    c_ip string,
    s_ip string,
    cs_method string,
    cs_uri_stem string,
    sc_status string,
    sc_bytes string,
    cs_bytes string,
    time_taken string,
    cs_version string
)
```
ROW FORMAT DELIMITED
  FIELDS TERMINATED BY '\''
STORED AS INPUTFORMAT
  'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION 's3://bucket-name/w3c-log-folder/'

3. Run the query in the Athena console to register the `iis_w3c_logs` table. When the query completes, the logs are ready for you to query from Athena.

Example W3C extended log select query

The following example query selects the date, time, request target, and time taken for the request from the table `iis_w3c_logs`. The WHERE clause filters for cases in which the HTTP method is GET and the HTTP status code is 200 (successful).

```
SELECT date_col, time_col, cs_uri_stem, time_taken
FROM iis_w3c_logs
WHERE cs_method = 'GET' AND sc_status = '200'
```

The following image shows the results of the query in the Athena Query Editor.

Combining the date and time fields

The space delimited date and time fields are separate entries in the log source data, but you can combine them into a timestamp if you want. Use the `concat()` and `date_parse()` functions in a `SELECT` (p. 692) or `CREATE TABLE AS SELECT` (p. 744) query to concatenate and convert the date and time columns into timestamp format. The following example uses a CTAS query to create a new table with a `derived_timestamp` column.

```
CREATE TABLE iis_w3c_logs_w_timestamp AS
SELECT
date_parse(concat(date_col,' ', time_col),'%Y-%m-%d %H:%i:%s') as derived_timestamp,
c_ip,
s_ip,
cs_method,
cs_uri_stem,
sc_status,
sc_bytes,
cs_bytes,
time_taken,
cs_version
FROM iis_w3c_logs
```
After the table is created, you can query the new timestamp column directly, as in the following example.

```
SELECT derived_timestamp, cs_uri_stem, time_taken
FROM iis_w3c_logs_w_timestamp
WHERE cs_method = 'GET' AND sc_status = '200'
```

The following image shows the results of the query.

![Query Results](image)

### IIS log file format

Unlike the W3C extended format, the IIS log file format has a fixed set of fields and includes a comma as a delimiter. The LazySimpleSerDe treats the comma as the delimiter and the space after the comma as the beginning of the next field.

The following example shows sample data in the IIS log file format.

```
203.0.113.15, -, 2020-02-24, 22:48:38, W3SVC2, SERVER5, 198.51.100.4, 254, 501, 488, 200, 0, GET, /index.htm, -,
203.0.113.4, -, 2020-02-24, 22:48:39, W3SVC2, SERVER6, 198.51.100.6, 147, 411, 388, 200, 0, GET, /about.html, -,
203.0.113.11, -, 2020-02-24, 22:48:40, W3SVC2, SERVER7, 198.51.100.18, 170, 531, 468, 200, 0, GET, /image.png, -,
203.0.113.8, -, 2020-02-24, 22:48:41, W3SVC2, SERVER8, 198.51.100.14, 125, 711, 868, 200, 0, GET, /intro.htm, -,
```

### Creating a table in Athena for IIS logs

To query your IIS log file format logs in Amazon S3, you first create a table schema so that Athena can read the log data.

**To create a table in Athena for IIS log file format logs**

2. Paste the following DDL statement into the Athena console, noting the following points:
   a. To specify the comma delimiter, use `FIELDS TERMINATED BY ','`.
   b. Modify the values in `LOCATION 's3://bucket-name/iis-log-file-folder/'` to point to your IIS log format log files in Amazon S3.

```
CREATE EXTERNAL TABLE 'iis_format_logs'(
  client_ip_address string,
  
```

494
user_name string,
request_date string,
request_time string,
service_and_instance string,
server_name string,
server_ip_address string,
time_taken_millisec string,
client_bytes_sent string,
service_bytes_sent string,
request_type string,
time_taken_millisec string
)

ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS INPUTFORMAT
'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION
's3://bucket-name/iis-log-file-folder/'

3. Run the query in the Athena console to register the iis_format_logs table. When the query completes, the logs are ready for you to query from Athena.

Example IIS log format select query

The following example query selects the request date, request time, request target, and time taken in milliseconds from the table iis_format_logs. The WHERE clause filters for cases in which the request type is 'GET' and the HTTP status code is 200 (successful). In the query, note that the leading spaces in 'GET' and ' 200' are required to make the query successful.

```
SELECT request_date, request_time, target_of_operation, time_taken_millisec
FROM iis_format_logs
WHERE request_type = ' GET' AND service_status_code = ' 200'
```

The following image shows the results of the query of the sample data.

NCSA log file format

IIS also uses the NCSA logging format, which has a fixed number of fields in ASCII text format separated by spaces. The structure is similar to the common log format used for Apache access logs. Fields in the NCSA common log data format include the client IP address, client ID (not typically used), domain\user ID, request received timestamp, text of the client request, server status code, and size of the object returned to the client.
Creating a table in Athena for IIS NCSA logs

For your CREATE TABLE statement, you can use the Grok SerDe (p. 317) and a grok pattern similar to the one for Apache web server logs (p. 490). Unlike Apache logs, the grok pattern uses %{DATA:user_id} for the third field instead of %{USERNAME:user_id} to account for the presence of the backslash in domain\user_id. For more information about using the Grok SerDe, see Writing grok custom classifiers in the AWS Glue Developer Guide.

To create a table in Athena for IIS NCSA web server logs

2. Paste the following DDL statement into the Athena Query Editor. Modify the values in LOCATION 's3://bucket-name/iis-ncsa-logs/' to point to your IIS NCSA logs in Amazon S3.

```sql
CREATE EXTERNAL TABLE iis_ncsa_logs(
    client_ip string,
    client_id string,
    user_id string,
    request_received_time string,
    client_request string,
    server_status string,
    returned_obj_size string
) ROW FORMAT SERDE 'com.amazonaws.glue.serde.GrokSerDe'
WITH SERDEPROPERTIES (
    'input.format'='^%{IPV4:client_ip} %{DATA:client_id} %{DATA:user_id}%{GREEDYDATA:request_received_time}%{QUOTEDSTRING:client_request}%{DATA:server_status} %{DATA:returned_obj_size}$'
) STORED AS INPUTFORMAT 'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION 's3://bucket-name/iis-ncsa-logs/';
```
3. Run the query in the Athena console to register the iis_ncsa_logs table. When the query completes, the logs are ready for you to query from Athena.
Example select queries for IIS NCSA logs

Example – Filtering for 404 errors

The following example query selects the request received time, text of the client request, and server status code from the iis_ncsa_logs table. The WHERE clause filters for HTTP status code 404 (page not found).

```
SELECT request_received_time, client_request, server_status
FROM iis_ncsa_logs
WHERE server_status = '404'
```

The following image shows the results of the query in the Athena Query Editor.

Example – Filtering for successful requests from a particular domain

The following example query selects the user ID, request received time, text of the client request, and server status code from the iis_ncsa_logs table. The WHERE clause filters for requests with HTTP status code 200 (successful) from users in the AnyCompany domain.

```
SELECT user_id, request_received_time, client_request, server_status
FROM iis_ncsa_logs
WHERE server_status = '200' AND user_id LIKE 'AnyCompany%'
```

The following image shows the results of the query in the Athena Query Editor.
Using Athena ACID transactions

The term "ACID transactions" refers to a set of properties (atomicity, consistency, isolation, and durability) that ensure data integrity in database transactions. ACID transactions enable multiple users to concurrently and reliably add and delete Amazon S3 objects in an atomic manner, while isolating any existing queries by maintaining read consistency for queries against the data lake. Athena ACID transactions add single-table support for insert, delete, update, and time travel operations to the Athena SQL data manipulation language (DML). You and multiple concurrent users can use Athena ACID transactions to make reliable, row-level modifications to Amazon S3 data. Athena transactions automatically manage locking semantics and coordination and do not require a custom record locking solution.

Athena ACID transactions and familiar SQL syntax simplify updates to your business and regulatory data. For example, to respond to a data erasure request, you can perform a SQL DELETE operation. To make manual record corrections, you can use a single UPDATE statement. To recover data that was recently deleted, you can issue time travel queries using a SELECT statement.

Because they are built on shared table formats, Athena ACID transactions are compatible with other services and engines such as Amazon EMR and Apache spark that also support shared table formats.

Athena transactions are available through the Athena console, API operations, and ODBC and JDBC drivers.

Topics
- Using governed tables (p. 498)
- Using Iceberg tables (p. 503)

Using governed tables

To help ensure secure and reliable transactions, Athena supports read operations using AWS Lake Formation governed tables. The security features help ensure that your users have access only to the data in your Amazon S3 based data lakes to which they’re authorized. The ACID features help ensure that queries are reliable in the face of complex changes to the underlying data.

Use Lake Formation data filtering to secure the tables in your Amazon S3 data lake by granting permissions at the column, row, and cell levels. These permissions are enforced when Athena users query your data. This fine level of control means that you can grant users access to sensitive information without using coarse-grained masking that would otherwise impede their analyses.

Governed tables in AWS Lake Formation provide the following capabilities:

- **ACID transactions** – Read and write to and from multiple tables in your Amazon S3 data lake using ACID (atomic, consistent, isolated, and durable) transactions. You can commit or cancel a transaction, or Lake Formation cancels the transaction if it finds an error. In Athena, governed table queries are read-only.

- **Time travel and version travel queries** – Each governed table maintains a versioned manifest of the Amazon S3 objects that it comprises. Previous versions of the manifest can be used for time travel
and version travel queries. You can perform time travel queries in Athena to query Amazon S3 for historical data as of a specified date and time. You can perform version travel queries in Athena to query Amazon S3 for historical data as of a specified version snapshot ID.

- **Enforce row-level permissions** – You can grant `SELECT` on a table and provide a predicate such as `region=us-west-1` and `year=2019`. Only table rows for which the predicate is true are returned. Row-level permissions are also available in Lake Formation for non-governed tables. For more information, see Data filtering and cell-level security in Lake Formation in the AWS Lake Formation Developer Guide.

- **Enforce cell-level permissions** – Cell-level security is a special case of row-level security that adds the ability to specify columns and rows in the same policy.

- **Enforce column-level permissions** – You can use AWS Lake Formation column-level permissions to restrict access to specific columns in a table. Column-level permissions are also available in Lake Formation for non-governed tables. For more information, see Data filtering and cell-level security in Lake Formation in the AWS Lake Formation Developer Guide.

- **Catalog transactions** – Athena supports creation, deletion, and schema updates of governed tables with transaction semantics. You can use this schema evolution support to query across schema or partition changes when the schema changes are forward compatible. To query earlier schemas that are not forward compatible, use time travel queries.

- **Automatic data compaction** – For improved performance, Lake Formation automatically compacts small Amazon S3 objects from governed tables into larger objects.

- **Predicate pushdown** – Predicate parts of SQL queries can be "pushed" closer to where the data lives. This optimization can drastically reduce query processing time by filtering out data earlier rather than later.

For more information about governed tables, see Managing governed tables in the AWS Lake Formation Developer Guide.

**Considerations and limitations**

Support for Lake Formation governed tables in Athena has the following limitations:

- **Read and create governed table operations only** – Only read and create governed table operations are supported. Delete, insert into, and update operations on Lake Formation governed tables are not supported from Athena.

- **Automatic compaction** – The Lake Formation automatic compaction feature is not manageable from Athena.

- **GetQueryStatistics permission required** – Athena users must add the `lakeformation:GetQueryStatistics` permission in IAM.

- **Error when querying certain nested columns on an empty dataset** – Currently, querying columns of the form `Array<Struct<Array<data_type>>>` on an empty dataset produces the error GENERIC_INTERNAL_ERROR: Exception while processing column.

- **Views** – You cannot use views with governed tables.

**Getting started**

To get started using Lake Formation governed tables in Athena, perform the following step:

- **Create a governed table** – To create a governed table from the Athena console, specify the table property `table_property='table_type'='LAKEFORMATION_GOVERNED'` in your `CREATE TABLE` statement. You can also use the AWS Lake Formation console, AWS Glue API, or AWS Command Line Interface (AWS CLI) to create one or more governed tables.
Creating governed tables

CREATE TABLE syntax

To create a governed table from Athena, set the table_type table property to LAKEFORMATION_GOVERNED in the TBL_PROPERTIES clause, as in the following syntax summary.

```
CREATE TABLE [db_name.]table_name (col_name data_type [COMMENT col_comment] [, ...] )
LOCATION 's3://DOC-EXAMPLE-BUCKET[/your-folder/]
TBLPROPERTIES ('table_type'='LAKEFORMATION_GOVERNED'
[, property_name=property_value] )
```

**Note**
The EXTERNAL keyword is not used in Athena to create governed tables. Using it results in the error message External keyword not supported for table type LAKEFORMATION_GOVERNED.

Example CREATE TABLE statement

The following example creates a governed table that has three columns.

```
CREATE TABLE governed_table (
  id int,
  data string,
  category string)
PARTITIONED BY (category)
LOCATION 's3://DOC-EXAMPLE-BUCKET/governed-folder'
TBLPROPERTIES ( 'table_type'='LAKEFORMATION_GOVERNED' )
```

Querying governed tables

To query a governed dataset, use a standard SELECT statement like the following.

```
SELECT * FROM databasename.tablename [WHERE predicate]
```

Predicate pushdown support

To optimize query times, some predicates are pushed down to where the data lives. The following tables summarize the Athena data types and logical operators currently supported in predicate pushdown.

<table>
<thead>
<tr>
<th>Athena data type</th>
<th>Pushed down</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>No</td>
</tr>
<tr>
<td>bigint</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### Athena data type

<table>
<thead>
<tr>
<th>Athena data type</th>
<th>Pushed down</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td>No</td>
</tr>
<tr>
<td>boolean</td>
<td>Yes</td>
</tr>
<tr>
<td>char</td>
<td>No</td>
</tr>
<tr>
<td>date</td>
<td>Yes</td>
</tr>
<tr>
<td>decimal</td>
<td>Yes</td>
</tr>
<tr>
<td>double</td>
<td>Yes</td>
</tr>
<tr>
<td>float</td>
<td>Yes</td>
</tr>
<tr>
<td>int, integer</td>
<td>Yes</td>
</tr>
<tr>
<td>map</td>
<td>No</td>
</tr>
<tr>
<td>smallint</td>
<td>Yes</td>
</tr>
<tr>
<td>string</td>
<td>Yes</td>
</tr>
<tr>
<td>struct</td>
<td>No</td>
</tr>
<tr>
<td>timestamp</td>
<td>Yes</td>
</tr>
<tr>
<td>tinyint</td>
<td>Yes</td>
</tr>
<tr>
<td>varchar</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Logical operator

<table>
<thead>
<tr>
<th>Logical operator</th>
<th>Example query</th>
<th>Pushed down</th>
</tr>
</thead>
<tbody>
<tr>
<td>single or multiple AND</td>
<td>SELECT * FROM table WHERE col_1 = a AND col_2 = b</td>
<td>Yes</td>
</tr>
<tr>
<td>OR conditions on same column</td>
<td>SELECT * FROM table WHERE col_1 = a OR col_1 = b</td>
<td>Yes</td>
</tr>
<tr>
<td>OR conditions on different columns</td>
<td>SELECT * FROM table WHERE col_1 = a OR col_2 = b</td>
<td>No</td>
</tr>
<tr>
<td>IS NULL, IS NOT NULL</td>
<td>SELECT * FROM table WHERE col_1 IS NOT NULL</td>
<td>Yes</td>
</tr>
<tr>
<td>NOT</td>
<td>SELECT * FROM table WHERE NOT col_1 = a</td>
<td>Yes</td>
</tr>
<tr>
<td>IN</td>
<td>SELECT * FROM table WHERE NOT col_1 IN (a, b)</td>
<td>Yes</td>
</tr>
<tr>
<td>BETWEEN</td>
<td>SELECT * FROM table WHERE col_1 BETWEEN A AND B</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Logical operator | Example query | Pushed down
--- | --- | ---
LIKE | SELECT * FROM table WHERE col_1 LIKE 'expression%' | No

### Time travel and version travel queries

Each governed table maintains a versioned manifest of the Amazon S3 objects that it comprises. Previous versions of the manifest can be used for time travel and version travel queries.

Time travel queries in Athena query Amazon S3 for historical data from a consistent snapshot as of a specified date and time. Version travel queries in Athena query Amazon S3 for historical data as of a specified snapshot ID.

#### Time travel queries

To run a time travel query, use `FOR SYSTEM_TIME AS OF timestamp` after the table name in the `SELECT` statement, as in the following example:

```sql
SELECT * FROM database.table FOR SYSTEM_TIME AS OF timestamp
```

The system time to be specified for traveling is either a timestamp or timestamp with a time zone. If not specified, Athena considers the value to be a timestamp in UTC time.

The following example time travel queries select data for the specified date and time.

```sql
SELECT * FROM governed_table
FOR SYSTEM_TIME AS OF TIMESTAMP '2020-01-01 10:00:00'
```

```sql
SELECT * FROM governed_table
FOR SYSTEM_TIME AS OF (current_timestamp - interval '1' day)
```

#### Version travel queries

To perform a version travel query (that is, view a consistent snapshot as of a specified version), use `FOR SYSTEM_VERSION AS OF version` after the table name in the `SELECT` statement, as in the following example.

```sql
SELECT * FROM database.table FOR SYSTEM_VERSION AS OF version
```

The `version` parameter is the bigint snapshot ID associated with a governed table version.

The following example version travel query selects data for the specified version.

```sql
SELECT * FROM governed_table FOR SYSTEM_VERSION AS OF 949530903748831860
```

#### Combining time and version travel

You can use time travel and version travel syntax in the same query to specify different timing and versioning conditions, as in the following example.

```sql
SELECT table1.*, table2.* FROM governed_table FOR SYSTEM_TIME AS OF (current_timestamp - interval '1' day) AS table1
```
### Supported data types

Governed tables currently support the following data types:

- array
- bigint
- boolean
- char
- date (YYYY-MM-DD)
- decimal
- double
- float
- integer
- long
- map
- smallint
- string
- struct
- timestamp
- varchar

**Note**

Currently, support for the double and float types in governed tables is limited. For float, Athena supports only up to 6 (instead of 9) decimals of precision. For double, Athena supports only up to 15 (instead of 17) digits of precision.

For more information about data types in Athena, see [Data types in Amazon Athena](p. 690).

### Using Iceberg tables

Athena supports read, time travel, write, and DDL queries for Apache Iceberg tables that use the Apache Parquet format for data and the AWS Glue catalog for their metastore.

**Apache Iceberg** is an open table format for very large analytic datasets. Iceberg manages large collections of files as tables, and it supports modern analytical data lake operations such as record-level insert, update, delete, and time travel queries. The Iceberg specification allows seamless table evolution such as schema and partition evolution, and its design is optimized for usage on Amazon S3. Iceberg also helps guarantee data correctness under concurrent write scenarios.

For more information about Apache Iceberg, see [https://iceberg.apache.org/](https://iceberg.apache.org/).

### Considerations and limitations

Athena support for Iceberg tables has the following limitations:

- **Tables with AWS Glue catalog only** – Only Iceberg tables created against the AWS Glue catalog based on specifications defined by the open source glue catalog implementation are supported from Athena.
- **Table locking support by AWS Glue only** – Unlike the open source Glue catalog implementation, which supports plug-in custom locking, Athena supports AWS Glue optimistic locking only. Using Athena to modify an Iceberg table with any other lock implementation will cause potential data loss and break transactions.
- **Supported file formats** – Iceberg file format support in Athena depends on the Athena engine version, as shown in the following table.

<table>
<thead>
<tr>
<th>Athena engine version</th>
<th>Parquet</th>
<th>ORC</th>
<th>Avro</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- **Iceberg v2 tables** – Athena only creates and operates on Iceberg v2 tables. For the difference between v1 and v2 tables, see Format version changes in the Apache Iceberg documentation.

- **Display of time types without time zone** – The time and timestamp without time zone types are displayed in UTC. If the time zone is unspecified in a filter expression on a time column, UTC is used.

- **Timestamp related data precision** – While Iceberg supports microsecond precision for the timestamp data type, Athena supports only millisecond precision for timestamps in both reads and writes. Athena only retains millisecond precision in time related columns for data that is rewritten during manual compaction operations.

- **Unsupported operations** – The following Athena operations are not supported for Iceberg tables.
  - ALTER TABLE SET LOCATION (p. 737)

- **Views** – Use CREATE VIEW to create Athena views as described in Working with views (p. 361). If you are interested in using the Iceberg view specification to create views, contact athena-feedback@amazon.com.

If you would like Athena to support a particular feature, send feedback to athena-feedback@amazon.com.

**Topics**
- Creating Iceberg tables (p. 504)
- Managing Iceberg tables (p. 507)
- Evolving Iceberg table schema (p. 510)
- Querying Iceberg table data and performing time travel (p. 511)
- Updating Iceberg table data (p. 513)
- Optimizing Iceberg tables (p. 515)
- Supported data types (p. 516)
- Other Athena operations on Iceberg tables (p. 517)

**Creating Iceberg tables**

Athena creates Iceberg v2 tables. For the difference between v1 and v2 tables, see Format version changes in the Apache Iceberg documentation.

Athena CREATE TABLE creates an Iceberg table with no data. You can query a table from external systems such as Apache Spark directly if the table uses the Iceberg open source glue catalog. You do not have to create an external table.

**Warning**
Running CREATE EXTERNAL TABLE results in the error message External keyword not supported for table type ICEBERG.

To create an Iceberg table from Athena, set the 'table_type' table property to 'ICEBERG' in the TBL_PROPERTIES clause, as in the following syntax summary.
Creating Iceberg tables

Partitioning

To create Iceberg tables with partitions, use PARTITIONED BY syntax. Columns used for partitioning must be specified in the columns declarations first. Within the PARTITIONED BY clause, the column type must not be included. You can also define partition transforms in CREATE TABLE syntax. To specify multiple columns for partitioning, separate the columns with the comma (,) character, as in the following example.

```
CREATE TABLE iceberg_table (id bigint, data string, category string)
PARTITIONED BY (category, bucket(16, id))
LOCATION 's3://DOC-EXAMPLE-BUCKET/your-folder/'
TBLPROPERTIES ( 'table_type' = 'ICEBERG' )
```

The following table shows the available partition transform functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Supported types</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>year(ts)</code></td>
<td>Partition by year</td>
<td>date, timestamp</td>
</tr>
<tr>
<td><code>month(ts)</code></td>
<td>Partition by month</td>
<td>date, timestamp</td>
</tr>
<tr>
<td><code>day(ts)</code></td>
<td>Partition by day</td>
<td>date, timestamp</td>
</tr>
<tr>
<td><code>hour(ts)</code></td>
<td>Partition by hour</td>
<td>timestamp</td>
</tr>
<tr>
<td><code>bucket(N, col)</code></td>
<td>Partition by hashed value mod N buckets. This is the same concept as hash bucketing for Hive tables.</td>
<td>int, long, decimal, date, timestamp, string, binary</td>
</tr>
<tr>
<td><code>truncate(L, col)</code></td>
<td>Partition by value truncated to L</td>
<td>int, long, decimal, string</td>
</tr>
</tbody>
</table>

Athena supports Iceberg's hidden partitioning. For more information, see Iceberg's hidden partitioning in the Apache Iceberg documentation.

Table properties

This section describes table properties that you can specify as key-value pairs in the TBLPROPERTIES clause of the CREATE TABLE statement. Athena allows only a predefined list of key-value pairs in the table properties for creating or altering Iceberg tables. The following tables show the table properties that you can specify. For more information about the compaction options, see Optimizing Iceberg tables (p. 515) in this documentation. If you would like Athena to support a specific open source table configuration property, send feedback to athena-feedback@amazon.com.

**Format**

<table>
<thead>
<tr>
<th>Description</th>
<th>Allowed property values</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>File data format</td>
<td></td>
</tr>
<tr>
<td>Allowed property values</td>
<td>parquet</td>
<td></td>
</tr>
<tr>
<td>Default value</td>
<td>parquet</td>
<td></td>
</tr>
</tbody>
</table>
**write_compression**

<table>
<thead>
<tr>
<th>Description</th>
<th>File compression codec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allowed values</td>
<td>gzip, snappy, zstd</td>
</tr>
<tr>
<td>Default value</td>
<td>gzip</td>
</tr>
</tbody>
</table>

**write_target_data_file_size_bytes**

<table>
<thead>
<tr>
<th>Description</th>
<th>Specifies the target size in bytes of the files produced by Athena.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allowed values</td>
<td>A positive number</td>
</tr>
<tr>
<td>Default value</td>
<td>536870912 (512 MB)</td>
</tr>
</tbody>
</table>

**optimize_rewrite_min_data_file_size_bytes**

<table>
<thead>
<tr>
<th>Description</th>
<th>Data optimization specific configuration. Files smaller than the specified value are included for optimization.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allowed values</td>
<td>A non-negative number. Must be less than 10GB and less than the value of write_target_data_file_size_bytes.</td>
</tr>
<tr>
<td>Default value</td>
<td>0.75 times the value of write_target_data_file_size_bytes.</td>
</tr>
</tbody>
</table>

**optimize_rewrite_max_data_file_size_bytes**

<table>
<thead>
<tr>
<th>Description</th>
<th>Data optimization specific configuration. Files larger than the specified value are included for optimization.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allowed values</td>
<td>A positive number. Must be less than 10GB and greater than the value of write_target_data_file_size_bytes.</td>
</tr>
<tr>
<td>Default value</td>
<td>1.8 times the value of write_target_data_file_size_bytes.</td>
</tr>
</tbody>
</table>

**optimize_rewrite_data_file_threshold**

<table>
<thead>
<tr>
<th>Description</th>
<th>Data optimization specific configuration. If there are fewer data files that require optimization than the given threshold, the files are not rewritten. This allows the accumulation of more data files to produce files closer to the target size and skip unnecessary computation for cost saving.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allowed values</td>
<td>A positive number. Must be less than 50.</td>
</tr>
<tr>
<td>Default value</td>
<td>5</td>
</tr>
</tbody>
</table>

**optimize_rewrite_delete_file_threshold**
Managing Iceberg tables

Managing Iceberg tables

Athena supports the following table DDL operations for Iceberg tables.

ALTER TABLE RENAME

Renames a table.
Because the table metadata of an Iceberg table is stored in Amazon S3, you can update the database and table name of an Iceberg managed table without affecting underlying table information.

**Synopsis**

```
ALTER TABLE [db_name.]table_name RENAME TO [new_db_name.]new_table_name
```

**Example**

```
ALTER TABLE my_db.my_table RENAME TO my_db2.my_table2
```

**ALTER TABLE SET PROPERTIES**

Adds properties to an Iceberg table and sets their assigned values.

In accordance with Iceberg specifications, table properties are stored in the Iceberg table metadata file rather than in AWS Glue. Athena does not accept custom table properties. Refer to the Table properties (p. 505) section for allowed key-value pairs. If you would like Athena to support a specific open source table configuration property, send feedback to athena-feedback@amazon.com.

**Synopsis**

```
ALTER TABLE [db_name.]table_name SET TBLPROPERTIES ('property_name' = 'property_value' [ , ... ])
```

**Example**

```
ALTER TABLE iceberg_table SET TBLPROPERTIES ('write_target_data_file_size_bytes'='536870912', 'optimize_rewrite_delete_file_threshold'='10')
```

**ALTER TABLE UNSET PROPERTIES**

Drops existing properties from an Iceberg table.

**Synopsis**

```
ALTER TABLE [db_name.]table_name UNSET TBLPROPERTIES ('property_name' [ , ... ])
```

**Example**

```
ALTER TABLE iceberg_table UNSET TBLPROPERTIES ('write_target_data_file_size_bytes')
```

**DESCRIBE TABLE**

Describes table information.

**Synopsis**

```
DESCRIBE [FORMATTED] [db_name.]table_name
```
When the FORMATTED option is specified, the output displays additional information such as table location and properties.

### Example

```
DESCRIBE iceberg_table
```

### DROP TABLE

Drops an Iceberg table.

**Warning**
Because Iceberg tables are considered managed tables in Athena, dropping an Iceberg table also removes all the data in the table.

**Synopsis**

```
DROP TABLE [IF EXISTS] [db_name.]table_name
```

### Example

```
DROP TABLE iceberg_table
```

### SHOW CREATE TABLE

Displays a CREATE TABLE DDL statement that can be used to recreate the Iceberg table in Athena. If Athena cannot reproduce the table structure (for example, because custom table properties are specified in the table), an UNSUPPORTED error is thrown.

**Synopsis**

```
SHOW CREATE TABLE [db_name.]table_name
```

### Example

```
SHOW CREATE TABLE iceberg_table
```

### SHOW TABLE PROPERTIES

Shows one or more table properties of an Iceberg table. Only Athena-supported table properties are shown.

**Synopsis**

```
SHOW TBLPROPERTIES [db_name.]table_name [("property_name")]
```

### Example

```
SHOW TBLPROPERTIES iceberg_table
```
Evolving Iceberg table schema

Iceberg schema updates are metadata-only changes. No data files are changed when you perform a schema update.

The Iceberg format supports the following schema evolution changes:

- **Add** – Adds a new column to a table or to a nested struct.
- **Drop** – Removes an existing column from a table or nested struct.
- **Rename** – Renames an existing column or field in a nested struct.
- **Reorder** – Changes the order of columns.
- **Type promotion** – Widens the type of a column, struct field, map key, map value, or list element. Currently, the following cases are supported for Iceberg tables:
  - integer to big integer
  - float to double
  - increasing the precision of a decimal type

**ALTER TABLE ADD COLUMNS**

Adds one or more columns to an existing Iceberg table.

**Synopsis**

```
ALTER TABLE [db_name.]table_name ADD COLUMNS (col_name data_type [, ...])
```

**Examples**

The following example adds a comment column of type string to an Iceberg table.

```
ALTER TABLE iceberg_table ADD COLUMNS (comment string)
```

The following example adds a point column of type struct to an Iceberg table.

```
ALTER TABLE iceberg_table
ADD COLUMNS (point struct<x: double, y: double>)
```

The following example adds a points column that is an array of structs to an Iceberg table.

```
ALTER TABLE iceberg_table
ADD COLUMNS (points array<struct<x: double, y: double>>>)
```

**ALTER TABLE DROP COLUMN**

Drops a column from an existing Iceberg table.

**Synopsis**

```
ALTER TABLE [db_name.]table_name DROP COLUMN col_name
```
Example

ALTER TABLE iceberg_table DROP COLUMN userid

**ALTER TABLE CHANGE COLUMN**

Changes the name, type, order or comment of a column.

**Note**
ALTER TABLE REPLACE COLUMNS is not supported. Because REPLACE COLUMNS removes all columns and then adds new ones, it is not supported for Iceberg. CHANGE COLUMN is the preferred syntax for schema evolution.

**Synopsis**

```
ALTER TABLE [db_name.]table_name
  CHANGE [COLUMN] col_old_name col_new_name column_type
  [COMMENT col_comment] [FIRST|AFTER column_name]
```

Example

ALTER TABLE iceberg_table CHANGE comment blog_comment string AFTER id

**SHOW COLUMNS**

Shows the columns in a table.

**Synopsis**

```
SHOW COLUMNS (FROM|IN) [db_name.]table_name
```

Example

SHOW COLUMNS FROM iceberg_table

**Querying Iceberg table data and performing time travel**

To query an Iceberg dataset, use a standard SELECT statement like the following. Queries follow the Apache Iceberg [format v2 spec](https://iceberg.apache.org/docs/format/) and perform merge-on-read of both position and equality deletes.

```
SELECT * FROM [db_name.]table_name [WHERE predicate]
```

To optimize query times, all predicates are pushed down to where the data lives.

**Time travel and version travel queries**

Each Apache Iceberg table maintains a versioned manifest of the Amazon S3 objects that it contains. Previous versions of the manifest can be used for time travel and version travel queries.
Time travel queries in Athena query Amazon S3 for historical data from a consistent snapshot as of a specified date and time. Version travel queries in Athena query Amazon S3 for historical data as of a specified snapshot ID.

**Time travel queries**

To run a time travel query, use `FOR TIMESTAMP AS OF timestamp` after the table name in the `SELECT` statement, as in the following example.

```sql
SELECT * FROM iceberg_table FOR TIMESTAMP AS OF timestamp
```

The system time to be specified for traveling is either a timestamp or timestamp with a time zone. If not specified, Athena considers the value to be a timestamp in UTC time.

The following example time travel queries select CloudTrail data for the specified date and time.

```sql
SELECT * FROM iceberg_table FOR TIMESTAMP AS OF TIMESTAMP '2020-01-01 10:00:00 UTC'
```

```sql
SELECT * FROM iceberg_table FOR TIMESTAMP AS OF (current_timestamp - interval '1' day)
```

**Version travel queries**

To execute a version travel query (that is, view a consistent snapshot as of a specified version), use `FOR VERSION AS OF version` after the table name in the `SELECT` statement, as in the following example.

```sql
SELECT * FROM [db_name.]
table_name FOR VERSION AS OF version
```

The `version` parameter is the bigint snapshot ID associated with an Iceberg table version.

The following example version travel query selects data for the specified version.

```sql
SELECT * FROM iceberg_table FOR VERSION AS OF 949530903748831860
```

**Note**

The `FOR SYSTEM_TIME AS OF` and `FOR SYSTEM_VERSION AS OF` clauses in Athena engine version 2 have been replaced by the `FOR TIMESTAMP AS OF` and `FOR VERSION AS OF` clauses in Athena engine version 3.

**Retrieving the snapshot ID**

You can use the SDKs provided by Iceberg in [Java](https://iceberg.apache.org/) and [Python](https://iceberg.apache.org/python/) to retrieve the Iceberg snapshot ID. The following example is in Java.

```java
import org.apache.iceberg.Table;
import org.apache.iceberg.aws.glue.GlueCatalog;
import org.apache.iceberg.catalog.TableIdentifier;
import org.apache.iceberg.util.SnapshotUtil;
import java.text.SimpleDateFormat;
import java.util.Date;

Catalog catalog = new GlueCatalog();
Map<String, String> properties = new HashMap<String, String>();
properties.put("warehouse", "s3://my-bucket/my-folder");
```
Updating Iceberg table data

catalog.initialize("my_catalog", properties);

Date date = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss").parse("2022/01/01 00:00:00");
long millis = date.getTime();

TableIdentifier name = TableIdentifier.of("db", "table");
Table table = catalog.loadTable(name);
long oldestSnapshotIdAfter2022 = SnapshotUtil.oldestAncestorAfter(table, millis);

Combining time and version travel

You can use time travel and version travel syntax in the same query to specify different timing and
versioning conditions, as in the following example.

```
SELECT table1.*, table2.* FROM
  [db_name.]table_name FOR TIMESTAMP AS OF (current_timestamp - interval '1' day) AS table1
  FULL JOIN
  [db_name.]table_name FOR VERSION AS OF 5487432386996890161 AS table2
ON table1.ts = table2.ts
WHERE (table1.id IS NULL OR table2.id IS NULL)
```

Creating and querying views with Iceberg tables

To create and query Athena views on Iceberg tables, use CREATE VIEW views as described in Working
with views (p. 361).

Example:

```
CREATE VIEW view1 AS SELECT * FROM iceberg_table
```

```
SELECT * FROM view1
```

If you are interested in using the Iceberg view specification to create views, contact athena-
feedback@amazon.com.

Working with Lake Formation fine-grained access control

Athena engine version 3 supports Lake Formation fine-grained access control with Iceberg tables,
including column level and row level security access control. This access control works with time
travel queries and with tables that have performed schema evolution. For more information, see Lake
Formation fine-grained access control and Athena workgroups (p. 584).

If you created your Iceberg table outside of Athena, use Apache Iceberg SDK version 0.13.0 or higher so
that your Iceberg table column information is populated in the AWS Glue Data Catalog. If your Iceberg
table does not contain column information in AWS Glue, you can use the Athena ALTER TABLE SET
PROPERTIES (p. 508) statement or the latest Iceberg SDK to fix the table and update the column
information in AWS Glue.

Updating Iceberg table data

Iceberg table data can be managed directly on Athena using INSERT, UPDATE, and DELETE queries. Each
data management transaction produces a new snapshot, which can be queried using time travel. The
UPDATE and DELETE statements follow the Iceberg format v2 row-level position delete specification and
enforce snapshot isolation.
Use the following commands to perform data management operations on Iceberg tables.

**INSERT INTO**

Inserts data into an Iceberg table. Athena Iceberg INSERT INTO is charged the same as current INSERT INTO queries for external Hive tables by the amount of data scanned. To insert data into an Iceberg table, use the following syntax, where `query` can be either `VALUES (val1, val2, ...)` or `SELECT (col1, col2, ...) FROM [db_name.]table_name WHERE predicate`. For SQL syntax and semantic details, see [INSERT INTO](p. 697).

```sql
INSERT INTO [db_name.]table_name [(col1, col2, ...)] query
```

The following examples insert values into the table `iceberg_table`.

```sql
INSERT INTO iceberg_table VALUES (1,'a','c1')
```

```sql
INSERT INTO iceberg_table (col1, col2, ...) VALUES (val1, val2, ...)
```

```sql
INSERT INTO iceberg_table SELECT * FROM another_table
```

**DELETE**

Athena Iceberg DELETE writes Iceberg position delete files to a table. This is known as a *merge-on-read* delete. In contrast to a *copy-on-write* delete, a merge-on-read delete is more efficient because it does not rewrite file data. When Athena reads Iceberg data, it merges the Iceberg position delete files with data files to produce the latest view of a table. To remove these position delete files, you can run the **REWRITE DATA compaction action** (p. 515). DELETE operations are charged by the amount of data scanned. For syntax, see [DELETE](p. 700).

The following example deletes rows from `iceberg_table` that have `c3` as the value for `category`.

```sql
DELETE FROM iceberg_table WHERE category='c3'
```

**UPDATE**

Athena Iceberg UPDATE writes Iceberg position delete files and newly updated rows as data files in the same transaction. UPDATE can be imagined as a combination of INSERT INTO and DELETE. UPDATE operations are charged by the amount of data scanned. For syntax, see [UPDATE](p. 700).

The following example updates the specified values in the table `iceberg_table`.

```sql
UPDATE iceberg_table SET category='c2' WHERE category='c1'
```

**MERGE INTO**

Conditionally updates, deletes, or inserts rows into an Iceberg table. A single statement can combine update, delete, and insert actions. For syntax, see [MERGE INTO](p. 701).

The following example deletes all customers from table `t` that are in the source table `s`.

```sql
MERGE INTO accounts t USING monthly_accounts_update s
ON t.customer = s.customer
```
WHEN MATCHED
THEN DELETE

The following example updates target table \( t \) with customer information from source table \( s \). For customer rows in table \( t \) that have matching customer rows in table \( s \), the example increments the purchases in table \( t \). If table \( t \) has no match for a customer row in table \( s \), the example inserts the customer row from table \( s \) into table \( t \).

\[
\text{MERGE INTO accounts } t \text{ USING monthly_accounts_update } s \\
\quad \text{ON (t.customer = s.customer)} \\
\quad \text{WHEN MATCHED} \\
\quad \quad \text{THEN UPDATE SET purchases = s.purchases + t.purchases} \\
\quad \quad \text{WHEN NOT MATCHED} \\
\quad \quad \quad \text{THEN INSERT (customer, purchases, address)} \\
\quad \quad \quad \quad \text{VALUES(s.customer, s.purchases, s.address)}
\]

The following example conditionally updates target table \( t \) with information from the source table \( s \). The example deletes any matching target row for which the source address is Centreville. For all other matching rows, the example adds the source purchases and sets the target address to the source address. If there is no match in the target table, the example inserts the row from the source table.

\[
\text{MERGE INTO accounts } t \text{ USING monthly_accounts_update } s \\
\quad \text{ON (t.customer = s.customer)} \\
\quad \text{WHEN MATCHED AND s.address = 'Centreville'} \\
\quad \quad \text{THEN DELETE} \\
\quad \text{WHEN MATCHED} \\
\quad \quad \text{THEN UPDATE} \\
\quad \quad \quad \text{SET purchases = s.purchases + t.purchases, address = s.address} \\
\quad \quad \text{WHEN NOT MATCHED} \\
\quad \quad \quad \text{THEN INSERT (customer, purchases, address)} \\
\quad \quad \quad \quad \text{VALUES(s.customer, s.purchases, s.address)}
\]

Optimizing Iceberg tables

As data accumulates into an Iceberg table, queries gradually become less efficient because of the increased processing time required to open files. Additional computational cost is incurred if the table contains delete files. In Iceberg, delete files store row-level deletes, and the engine must apply the deleted rows to query results.

To help optimize the performance of queries on Iceberg tables, Athena supports manual compaction as a table maintenance command. Compactions optimize the structural layout of the table without altering table content.

**OPTIMIZE**

The OPTIMIZE `table` REWRITE DATA compaction action rewrites data files into a more optimized layout based on their size and number of associated delete files. For syntax and table property details, see OPTIMIZE (p. 701).

Example

The following example merges delete files into data files and produces files near the targeted file size where the value of catalog is c1.

\[
\text{OPTIMIZE iceberg_table REWRITE DATA USING BIN_PACK} \\
\quad \text{WHERE catalog = 'c1'}
\]
VACUUM

VACUUM performs snapshot expiration and orphan file removal. These actions reduce metadata size and remove files not in the current table state that are also older than the retention period specified for the table. For syntax details, see VACUUM (p. 702).

Example

The following example uses a table property to configure the table iceberg_table to retain the last three days of data, then uses VACUUM to expire the old snapshots and remove the orphan files from the table.

```
ALTER TABLE iceberg_table SET TBLPROPERTIES (  
    'vacuum_max_snapshot_age_seconds'='259200'  
)
VACUUM iceberg_table
```

Supported data types

Athena can query Iceberg tables that contain the following data types:

- binary
- boolean
- date
- decimal
- double
- float
- int
- list
- long
- map
- string
- struct
- timestamp without time zone

For more information about Iceberg table types, see the schemas page for Iceberg in the Apache documentation.

The following table shows the relationship between Athena data types and Iceberg table data types:

<table>
<thead>
<tr>
<th>Iceberg type</th>
<th>Athena type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>tinyint</td>
<td>Not supported for Iceberg tables in Athena.</td>
</tr>
<tr>
<td>-</td>
<td>smallint</td>
<td>Not supported for Iceberg tables in Athena.</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>In Athena DML statements, this type is INTEGER.</td>
</tr>
<tr>
<td>long</td>
<td>bigint</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td></td>
</tr>
</tbody>
</table>
### Iceberg type | Athena type | Notes
--- | --- | ---
decimal(P, S) | decimal(P, S) | S is precision, S is scale.
- | char | Not supported for Iceberg tables in Athena.
string | string | In Athena DML statements, this type is VARCHAR.
binary | binary | 
date | date | 
time | - | Only Iceberg timestamp (without time zone) is supported for Athena Iceberg DDL statements like CREATE TABLE, but all timestamp types can be queried through Athena.
timestamp | timestamp | 
timestamptz | timestamptz | 
list<E> | array | 
map<K,V> | map | 
struct<...> | struct | 
fixed(L) | - | The fixed(L) type is not currently supported in Athena.

For more information about data types in Athena, see Data types in Amazon Athena (p. 690).

### Other Athena operations on Iceberg tables

#### Database level operations

When you use DROP DATABASE (p. 754) with the CASCADE option, any Iceberg table data is also removed. The following DDL operations have no effect on Iceberg tables.

- CREATE DATABASE (p. 739)
- ALTER DATABASE SET DBPROPERTIES (p. 732)
- SHOW DATABASES (p. 760)
- SHOW TABLES (p. 762)
- SHOW VIEWS (p. 764)

#### Partition related operations

Because Iceberg tables use hidden partitioning, you do not have to work with physical partitions directly. As a result, Iceberg tables in Athena do not support the following partition-related DDL operations:

- SHOW PARTITIONS (p. 760)
- ALTER TABLE ADD PARTITION (p. 733)
- ALTER TABLE DROP PARTITION (p. 735)
- ALTER TABLE RENAME PARTITION (p. 735)

If you would like to see Iceberg partition evolution in Athena, send feedback to athena-feedback@amazon.com.
Unloading Iceberg tables

Iceberg tables can be unloaded to files in a folder on Amazon S3. For information, see UNLOAD (p. 717).

MSCK REPAIR

Because Iceberg tables keep track of table layout information, running MSCK REPAIR TABLE (p. 756) as one does with Hive tables is not necessary and is not supported.
Amazon Athena security

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security of the cloud and security in the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. The effectiveness of our security is regularly tested and verified by third-party auditors as part of the AWS compliance programs. To learn about the compliance programs that apply to Athena, see AWS services in scope by compliance program.

- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your organization’s requirements, and applicable laws and regulations.

This documentation will help you understand how to apply the shared responsibility model when using Amazon Athena. The following topics show you how to configure Athena to meet your security and compliance objectives. You'll also learn how to use other AWS services that can help you to monitor and secure your Athena resources.

**Topics**
- Data protection in Athena (p. 519)
- Identity and access management in Athena (p. 530)
- Logging and monitoring in Athena (p. 574)
- Compliance validation for Amazon Athena (p. 577)
- Resilience in Athena (p. 578)
- Infrastructure security in Athena (p. 578)
- Configuration and vulnerability analysis in Athena (p. 580)
- Using Athena to query data registered with AWS Lake Formation (p. 580)

Data protection in Athena

The AWS shared responsibility model applies to data protection in Amazon Athena. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. This content includes the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the AWS Security Blog.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual user accounts with AWS Identity and Access Management (IAM). That way each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We recommend TLS 1.2 or later.
• Set up API and user activity logging with AWS CloudTrail.
• Use AWS encryption solutions, along with all default security controls within AWS services.
• Use advanced managed security services such as Amazon Macie, which assists in discovering and securing personal data that is stored in Amazon S3.
• If you require FIPS 140-2 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-2.

We strongly recommend that you never put confidential or sensitive information, such as your customers’ email addresses, into tags or free-form fields such as a Name field. This includes when you work with Athena or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

As an additional security step, you can use the aws:CalledVia global condition context key to limit requests to only those made from Athena. For more information, see Using Athena with CalledVia context keys (p. 555).

Protecting multiple types of data

Multiple types of data are involved when you use Athena to create databases and tables. These data types include source data stored in Amazon S3, metadata for databases and tables that you create when you run queries or the AWS Glue Crawler to discover data, query results data, and query history. This section discusses each type of data and provides guidance about protecting it.

• Source data – You store the data for databases and tables in Amazon S3, and Athena does not modify it. For more information, see Data protection in Amazon S3 in the Amazon Simple Storage Service User Guide. You control access to your source data and can encrypt it in Amazon S3. You can use Athena to create tables based on encrypted datasets in Amazon S3 (p. 526).

• Database and table metadata (schema) – Athena uses schema-on-read technology, which means that your table definitions are applied to your data in Amazon S3 when Athena runs queries. Any schemas you define are automatically saved unless you explicitly delete them. In Athena, you can modify the Data Catalog metadata using DDL statements. You can also delete table definitions and schema without impacting the underlying data stored in Amazon S3.

  Note
  The metadata for databases and tables you use in Athena is stored in the AWS Glue Data Catalog. We highly recommend that you upgrade (p. 31) to using the AWS Glue Data Catalog with Athena. For more information about the benefits of using the AWS Glue Data Catalog, see FAQ: Upgrading to the AWS Glue Data Catalog (p. 33).

You can define fine-grained access policies to databases and tables (p. 540) registered in the AWS Glue Data Catalog using AWS Identity and Access Management (IAM). You can also encrypt metadata in the AWS Glue Data Catalog. If you encrypt the metadata, use permissions to encrypted metadata (p. 523) for access.

• Query results and query history, including saved queries – Query results are stored in a location in Amazon S3 that you can choose to specify globally, or for each workgroup. If not specified, Athena uses the default location in each case. You control access to Amazon S3 buckets where you store query results and saved queries. Additionally, you can choose to encrypt query results that you store in Amazon S3. Your users must have the appropriate permissions to access the Amazon S3 locations and decrypt files. For more information, see Encrypting Athena query results stored in Amazon S3 (p. 523) in this document.

Athena retains query history for 45 days. You can view query history (p. 349) using Athena APIs, in the console, and with AWS CLI. To store the queries for longer than 45 days, save them. To protect access
to saved queries, use workgroups (p. 629) in Athena, restricting access to saved queries only to users who are authorized to view them.

Topics

- Encryption at rest (p. 521)
- Encryption in transit (p. 529)
- Key management (p. 529)
- Internetwork traffic privacy (p. 529)

Encryption at rest

You can run queries in Amazon Athena on encrypted data in Amazon S3 in the same Region and across a limited number of Regions. You can also encrypt the query results in Amazon S3 and the data in the AWS Glue Data Catalog.

You can encrypt the following assets in Athena:

- The results of all queries in Amazon S3, which Athena stores in a location known as the Amazon S3 results location. You can encrypt query results stored in Amazon S3 whether the underlying dataset is encrypted in Amazon S3 or not. For information, see Encrypting Athena query results stored in Amazon S3 (p. 523).
- The data in the AWS Glue Data Catalog. For information, see Permissions to encrypted metadata in the AWS Glue Data Catalog (p. 523).

Note

The setup for querying an encrypted dataset in Amazon S3 and the options in Athena to encrypt query results are independent. Each option is enabled and configured separately. You can use different encryption methods or keys for each. This means that reading encrypted data in Amazon S3 doesn't automatically encrypt Athena query results in Amazon S3. The opposite is also true. Encrypting Athena query results in Amazon S3 doesn't encrypt the underlying dataset in Amazon S3.

Topics

- Supported Amazon S3 encryption options (p. 521)
- Permissions to encrypted data in Amazon S3 (p. 522)
- Permissions to encrypted metadata in the AWS Glue Data Catalog (p. 523)
- Encrypting Athena query results stored in Amazon S3 (p. 523)
- Creating tables based on encrypted datasets in Amazon S3 (p. 526)

Supported Amazon S3 encryption options

Athena supports the following encryption options for datasets and query results in Amazon S3.

<table>
<thead>
<tr>
<th>Encryption type</th>
<th>Description</th>
<th>Cross-Region support</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE-S3</td>
<td>Server side encryption (SSE) with an Amazon S3-managed key.</td>
<td>Yes</td>
</tr>
<tr>
<td>SSE-KMS</td>
<td>Server-side encryption (SSE) with a AWS Key Management Service customer managed key.</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Encryption at rest

<table>
<thead>
<tr>
<th>Encryption type</th>
<th>Description</th>
<th>Cross-Region support</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Note</strong></td>
<td>With this encryption type, Athena does not require you to indicate that data is encrypted when you create a table.</td>
<td>No</td>
</tr>
</tbody>
</table>

**CSE-KMS**  
Client-side encryption (CSE) with a AWS KMS customer managed key. In Athena, this option requires that you use a CREATE TABLE statement with a TBLPROPERTIES clause that specifies 'has_encrypted_data'='true'. For more information, see Creating tables based on encrypted datasets in Amazon S3 (p. 526).

For more information about AWS KMS encryption with Amazon S3, see What is AWS Key Management Service and How Amazon Simple Storage Service (Amazon S3) uses AWS KMS in the AWS Key Management Service Developer Guide. For more information about using SSE-KMS or CSE-KMS with Athena, see Launch: Amazon Athena adds support for querying encrypted data from the AWS Big Data Blog.

**Unsupported options**

The following encryption options are not supported:

- SSE with customer-provided keys (SSE-C).
- Client-side encryption using a client-side managed key.
- Asymmetric keys.

To compare Amazon S3 encryption options, see Protecting data using encryption in the Amazon Simple Storage Service User Guide.

**Tools for client-side encryption**

For client-side encryption, note that two tools are available:

- Amazon S3 encryption client – This encrypts data for Amazon S3 only and is supported by Athena.
- AWS Encryption SDK – The SDK can be used to encrypt data anywhere across AWS but is not directly supported by Athena.

These tools are not compatible, and data encrypted using one tool cannot be decrypted by the other. Athena only supports the Amazon S3 Encryption Client directly. If you use the SDK to encrypt your data, you can run queries from Athena, but the data is returned as encrypted text.

If you want to use Athena to query data that has been encrypted with the AWS Encryption SDK, you must download and decrypt your data, and then encrypt it again using the Amazon S3 Encryption Client.

**Permissions to encrypted data in Amazon S3**

Depending on the type of encryption you use in Amazon S3, you may need to add permissions, also known as "Allow" actions, to your policies used in Athena:

- **SSE-S3** – If you use SSE-S3 for encryption, Athena users require no additional permissions in their policies. It is sufficient to have the appropriate Amazon S3 permissions for the appropriate Amazon S3 location and for Athena actions. For more information about policies that allow appropriate Athena
and Amazon S3 permissions, see AWS managed policies for Amazon Athena (p. 530) and Access to Amazon S3 (p. 536).

- AWS KMS – If you use AWS KMS for encryption, Athena users must be allowed to perform particular AWS KMS actions in addition to Athena and Amazon S3 permissions. You allow these actions by editing the key policy for the AWS KMS customer managed CMKs that are used to encrypt data in Amazon S3. To add key users to the appropriate AWS KMS key policies, you can use the AWS KMS console at https://console.aws.amazon.com/kms. For information about how to add a user to a AWS KMS key policy, see Allows key users to use the CMK in the AWS Key Management Service Developer Guide.

  **Note**
  Advanced key policy administrators can adjust key policies. `kms:Decrypt` is the minimum allowed action for an Athena user to work with an encrypted dataset. To work with encrypted query results, the minimum allowed actions are `kms:GenerateDataKey` and `kms:Decrypt`.

When using Athena to query datasets in Amazon S3 with a large number of objects that are encrypted with AWS KMS, AWS KMS may throttle query results. This is more likely when there are a large number of small objects. Athena backs off retry requests, but a throttling error might still occur. If you are working with a large number of encrypted objects and experience this issue, one option is to enable Amazon S3 bucket keys to reduce the number of calls to KMS. For more information, see Reducing the cost of SSE-KMS with Amazon S3 Bucket keys in the Amazon Simple Storage Service User Guide. Another option is to increase your service quotas for AWS KMS. For more information, see Quotas in the AWS Key Management Service Developer Guide.

For troubleshooting information about permissions when using Amazon S3 with Athena, see the Permissions (p. 772) section of the Troubleshooting in Athena (p. 766) topic.

### Permissions to encrypted metadata in the AWS Glue Data Catalog

If you encrypt metadata in the AWS Glue Data Catalog, you must add "kms:GenerateDataKey", "kms:Decrypt", and "kms:Encrypt" actions to the policies you use for accessing Athena. For information, see Access from Athena to encrypted metadata in the AWS Glue Data Catalog (p. 553).

### Encrypting Athena query results stored in Amazon S3

You set up query result encryption using the Athena console or when using JDBC or ODBC. Workgroups allow you to enforce the encryption of query results.

In the console, you can configure the setting for encryption of query results in two ways:

- **Client-side settings** – When you use Settings in the console or the API operations to indicate that you want to encrypt query results, this is known as using client-side settings. Client-side settings include query results location and encryption. If you specify them, they are used, unless they are overridden by the workgroup settings.

- **Workgroup settings** – When you create or edit a workgroup (p. 640) and select the Override client-side settings field, then all queries that run in this workgroup use the workgroup encryption and query results location settings. For more information, see Workgroup settings override client-side settings (p. 638).

#### To encrypt query results stored in Amazon S3 using the console

**Important**
If your workgroup has the Override client-side settings field selected, then all queries in the workgroup use the workgroup settings. The encryption configuration and the query results
location specified on the Settings tab in the Athena console, by API operations and by JDBC and ODBC drivers are not used. For more information, see Workgroup settings override client-side settings (p. 638).

1. In the Athena console, choose Settings.

2. Choose Manage.

3. For Location of query result, enter or choose an Amazon S3 path. This is the Amazon S3 location where query results are stored.

4. Choose Encrypt query results.
Manage settings

Query result location and encryption

Location of query result

s3://bucket/prefix/object/

Encrypt query results

Encryption type
Choose server-side encryption (SSE) with an S3-managed encryption key (SSE-S3) or a customer master key (CMK) that you provide (SSE-KMS). Encryption with a CMK (CSE-KMS).

CSE_KMS

Choose an AWS KMS key
This key will be used to encrypt and decrypt your resources.
5. For **Encryption type**, choose **CSE-KMS**, **SSE-KMS**, or **SSE-S3**.
6. If you chose **SSE-KMS** or **CSE-KMS**, specify an AWS KMS key.
   - For **Choose an AWS KMS key**, if your account has access to an existing AWS KMS customer managed key (CMK), choose its alias or enter an AWS KMS key ARN.
   - If your account does not have access to an existing customer managed key (CMK), choose **Create an AWS KMS key**, and then open the AWS KMS console. For more information, see **Creating keys** in the **AWS Key Management Service Developer Guide**.

   **Note**
   Athena supports only symmetric keys for reading and writing data.
7. Return to the Athena console and choose the key that you created by alias or ARN.
8. Choose **Save**.

**Encrypting Athena query results when using JDBC or ODBC**

If you connect using the JDBC or ODBC driver, you configure driver options to specify the type of encryption to use and the Amazon S3 staging directory location. To configure the JDBC or ODBC driver to encrypt your query results using any of the encryption protocols that Athena supports, see **Connecting to Amazon Athena with ODBC and JDBC drivers** (p. 183).

**Creating tables based on encrypted datasets in Amazon S3**

When you create a table, indicate to Athena that a dataset is encrypted in Amazon S3. This is not required when using SSE-KMS. For both SSE-S3 and AWS KMS encryption, Athena determines how to decrypt the dataset and create the table, so you don’t need to provide key information.

Users that run queries, including the user who creates the table, must have the permissions described earlier in this topic.

**Important**
If you use Amazon EMR along with EMRFS to upload encrypted Parquet files, you must disable multipart uploads by setting `fs.s3n.multipart.uploads.enabled` to `false`. If you don’t do this, Athena is unable to determine the Parquet file length and a **HIVE_CANNOT_OPEN_SPLIT** error occurs. For more information, see **Configure multipart upload for Amazon S3** in the **Amazon EMR Management Guide**.

To indicate that the dataset is encrypted in Amazon S3, perform one of the following steps. This step is not required if SSE-KMS is used.

- In a **CREATE TABLE** (p. 740) statement, use a TBLPROPERTIES clause that specifies `'has_encrypted_data'='true'`, as in the following example.

```sql
CREATE EXTERNAL TABLE 'my_encrypted_data' (    'n_nationkey' int,    'n_name' string,    'n_regionkey' int,    'n_comment' string) ROW FORMAT SERDE    'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe' STORED AS INPUTFORMAT    'org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat' LOCATION    's3://bucket/folder_with_my_encrypted_data/' TBLPROPERTIES (    'has_encrypted_data'='true')
```

- Use the **JDBC driver** (p. 183) and set the TBLPROPERTIES value as shown in the previous example when you use `statement.executeQuery()` to run the **CREATE TABLE** (p. 740) statement.
When you use the Athena console to create a table using a form (p. 22) and specify the table location, select the **Encrypted data set** option.

In the Athena console list of tables, encrypted tables display a key-shaped icon.
Encryption in transit

In addition to encrypting data at rest in Amazon S3, Amazon Athena uses Transport Layer Security (TLS) encryption for data in-transit between Athena and Amazon S3, and between Athena and customer applications accessing it.

You should allow only encrypted connections over HTTPS (TLS) using the `aws:SecureTransport` condition on Amazon S3 bucket IAM policies.

Query results that stream to JDBC or ODBC clients are encrypted using TLS. For information about the latest versions of the JDBC and ODBC drivers and their documentation, see Using Athena with the JDBC driver (p. 183) and Connecting to Amazon Athena with ODBC (p. 185).

Key management

Amazon Athena supports AWS Key Management Service (AWS KMS) to encrypt datasets in Amazon S3 and Athena query results. AWS KMS uses customer managed keys (CMKs) to encrypt your Amazon S3 objects and relies on envelope encryption.

In AWS KMS, you can perform the following actions:

- Create keys
- Import your own key material for new CMKs

**Note**

Athena supports only symmetric keys for reading and writing data.

For more information, see What is AWS Key Management Service in the AWS Key Management Service Developer Guide, and How Amazon Simple Storage Service uses AWS KMS. To view the keys in your account that AWS creates and manages for you, in the navigation pane, choose AWS managed keys.

If you are uploading or accessing objects encrypted by SSE-KMS, use AWS Signature Version 4 for added security. For more information, see Specifying the signature version in request authentication in the Amazon Simple Storage Service User Guide.

If your Athena workloads encrypt a large amount of data, you can use Amazon S3 Bucket Keys to reduce costs. For more information, see Reducing the cost of SSE-KMS with Amazon S3 Bucket keys in the Amazon Simple Storage Service User Guide.

Internetwork traffic privacy

Traffic is protected both between Athena and on-premises applications and between Athena and Amazon S3. Traffic between Athena and other services, such as AWS Glue and AWS Key Management Service, uses HTTPS by default.

- **For traffic between Athena and on-premises clients and applications**, query results that stream to JDBC or ODBC clients are encrypted using Transport Layer Security (TLS).

  You can use one of the connectivity options between your private network and AWS:
  - A Site-to-Site VPN AWS VPN connection. For more information, see What is Site-to-Site VPN AWS VPN in the AWS Site-to-Site VPN User Guide.
  - An AWS Direct Connect connection. For more information, see What is AWS Direct Connect in the AWS Direct Connect User Guide.
  - **For traffic between Athena and Amazon S3 buckets**, Transport Layer Security (TLS) encrypts objects in-transit between Athena and Amazon S3, and between Athena and customer applications accessing it, you should allow only encrypted connections over HTTPS (TLS) using the `aws:SecureTransport` condition.
condition on Amazon S3 bucket IAM policies. Although Athena currently uses the public endpoint to access data in Amazon S3 buckets, this does not mean that the data traverses the public internet. All traffic between Athena and Amazon S3 is routed over the AWS network and is encrypted using TLS.

- **Compliance programs** – Amazon Athena complies with multiple AWS compliance programs, including SOC, PCI, FedRAMP, and others. For more information, see AWS services in scope by compliance program.

## Identity and access management in Athena

Amazon Athena uses AWS Identity and Access Management (IAM) policies to restrict access to Athena operations. For a full list of permissions for Athena, see Actions, resources, and condition keys for Amazon Athena in the Service Authorization Reference.

Whenever you use IAM policies, make sure that you follow IAM best practices. For more information, see Security best practices in IAM in the IAM User Guide.

The permissions required to run Athena queries include the following:

- Amazon S3 locations where the underlying data to query is stored. For more information, see Identity and access management in Amazon S3 in the Amazon Simple Storage Service User Guide.
- Metadata and resources that you store in the AWS Glue Data Catalog, such as databases and tables, including additional actions for encrypted metadata. For more information, see Setting up IAM permissions for AWS Glue and Setting up encryption in AWS Glue in the AWS Glue Developer Guide.
- Athena API actions. For a list of API actions in Athena, see Actions in the Amazon Athena API Reference.

The following topics provide more information about permissions for specific areas of Athena.

**Topics**

- AWS managed policies for Amazon Athena (p. 530)
- Access through JDBC and ODBC connections (p. 536)
- Access to Amazon S3 (p. 536)
- Cross-account access in Athena to Amazon S3 buckets (p. 537)
- Fine-grained access to databases and tables in the AWS Glue Data Catalog (p. 540)
- Cross-account access to AWS Glue data catalogs (p. 548)
- Access from Athena to encrypted metadata in the AWS Glue Data Catalog (p. 553)
- Access to workgroups and tags (p. 553)
- Allow access to prepared statements (p. 554)
- Using Athena with CalledVia context keys (p. 555)
- Allow access to an Athena Data Connector for External Hive Metastore (p. 557)
- Allow Lambda function access to external Hive metastores (p. 559)
- Example IAM permissions policies to allow Athena Federated Query (p. 562)
- Example IAM permissions policies to allow Amazon Athena User Defined Functions (UDF) (p. 566)
- Allowing access for ML with Athena (p. 570)
- Enabling federated access to the Athena API (p. 571)

## AWS managed policies for Amazon Athena

To add permissions to users, groups, and roles, it is easier to use AWS managed policies than to write policies yourself. It takes time and expertise to create IAM customer managed policies that provide your
team with only the permissions they need. To get started quickly, you can use our AWS managed policies. These policies cover common use cases and are available in your AWS account. For more information about AWS managed policies, see AWS managed policies in the IAM User Guide.

AWS services maintain and update AWS managed policies. You can't change the permissions in AWS managed policies. Services occasionally add additional permissions to an AWS managed policy to support new features. This type of update affects all identities (users, groups, and roles) where the policy is attached. Services are most likely to update an AWS managed policy when a new feature is launched or when new operations become available. Services do not remove permissions from an AWS managed policy, so policy updates won't break your existing permissions.

Additionally, AWS supports managed policies for job functions that span multiple services. For example, the ViewOnlyAccess AWS managed policy provides read-only access to many AWS services and resources. When a service launches a new feature, AWS adds read-only permissions for new operations and resources. For a list and descriptions of job function policies, see AWS managed policies for job functions in the IAM User Guide.

**Considerations when using managed policies with Athena**

Managed policies are easy to use and are updated automatically with the required actions as the service evolves. When using managed policies with Athena, keep the following points in mind:

- To allow or deny Amazon Athena service actions for yourself or other users using AWS Identity and Access Management (IAM), you attach identity-based policies to principals, such as users or groups.
- Each identity-based policy consists of statements that define the actions that are allowed or denied. For more information and step-by-step instructions for attaching a policy to a user, see Attaching managed policies in the IAM User Guide. For a list of actions, see the Amazon Athena API Reference.
- Customer-managed and inline identity-based policies allow you to specify more detailed Athena actions within a policy to fine-tune access. We recommend that you use the AmazonAthenaFullAccess policy as a starting point and then allow or deny specific actions listed in the Amazon Athena API Reference. For more information about inline policies, see Managed policies and inline policies in the IAM User Guide.
- If you also have principals that connect using JDBC, you must provide the JDBC driver credentials to your application. For more information, see Access through JDBC and ODBC connections (p. 536).
- If you have encrypted the AWS Glue Data Catalog, you must specify additional actions in the identity-based IAM policies for Athena. For more information, see Access from Athena to encrypted metadata in the AWS Glue Data Catalog (p. 553).
- If you create and use workgroups, make sure your policies include relevant access to workgroup actions. For detailed information, see the section called "IAM policies for accessing workgroups" (p. 632) and the section called "Workgroup example policies" (p. 633).

**AWS managed policy: AmazonAthenaFullAccess**

The AmazonAthenaFullAccess managed policy grants full access to Athena. Attach it to IAM users and other principals who need full access to Athena. You can attach the AmazonAthenaFullAccess policy to your IAM identities.

**Permissions groupings**

The AmazonAthenaFullAccess policy is grouped into the following sets of permissions.

- **athena** – Allows principals access to Athena resources.
- **glue** – Allows principals access to AWS Glue databases, tables, and partitions. This is required so that the principal can use the AWS Glue Data Catalog with Athena.
• **s3** – Allows the principal to write and read query results from Amazon S3, to read publically available Athena data examples that reside in Amazon S3, and to list buckets. This is required so that the principal can use Athena to work with Amazon S3.

• **sns** – Allows principals to list Amazon SNS topics and get topic attributes. This enables principals to use Amazon SNS topics with Athena for monitoring and alert purposes.

• **cloudwatch** – Allows principals to create, read, and delete CloudWatch alarms. For more information, see [Controlling costs and monitoring queries with CloudWatch metrics and events](p. 647).

• **lakeformation** – Allows principals to request temporary credentials to access data in a data lake location that is registered with Lake Formation. For more information, see [Underlying data access control](in the *AWS Lake Formation Developer Guide*).

```json
{
"Version": "2012-10-17",
"Statement": [
{
"Effect": "Allow",
"Action": [
"athena:*"
],
"Resource": ["
"*
]
},
{
"Effect": "Allow",
"Action": [
"glue:CreateDatabase",
"glue:DeleteDatabase",
"glue:GetDatabase",
"glue:GetDatabases",
"glue:UpdateDatabase",
"glue:CreateTable",
"glue:DeleteTable",
"glue:BatchDeleteTable",
"glue:UpdateTable",
"glue:GetTable",
"glue:GetTables",
"glue:BatchCreatePartition",
"glue:CreatePartition",
"glue:DeletePartition",
"glue:BatchDeletePartition",
"glue:UpdatePartition",
"glue:GetPartition",
"glue:GetPartitions",
"glue:BatchGetPartition"
],
"Resource": ["
"*
]
},
{
"Effect": "Allow",
"Action": [
"s3:GetBucketLocation",
"s3:GetObject",
"s3:ListBucket",
"s3:ListBucketMultipartUploads",
"s3:ListMultipartUploadParts",
"s3:AbortMultipartUpload",
"s3:CreateBucket",
"s3:PutObject",
```

---

532
"s3:PutBucketPublicAccessBlock"
},
"Resource": [
  "arn:aws:s3:::aws-athena-query-results-*"
]
},
{
  "Effect": "Allow",
  "Action": [
    "s3:GetObject",
    "s3:ListBucket"
  ],
  "Resource": [
    "arn:aws:s3:::athena-examples*"
  ]
},
{
  "Effect": "Allow",
  "Action": [
    "s3:ListBucket",
    "s3:GetBucketLocation",
    "s3:ListAllMyBuckets"
  ],
  "Resource": [
    "*
  ]
},
{
  "Effect": "Allow",
  "Action": [
    "sns:ListTopics",
    "sns:GetTopicAttributes"
  ],
  "Resource": [
    "*
  ]
},
{
  "Effect": "Allow",
  "Action": [
    "cloudwatch:PutMetricAlarm",
    "cloudwatch:DescribeAlarms",
    "cloudwatch:DeleteAlarms",
    "cloudwatch:GetMetricData"
  ],
  "Resource": [
    "*
  ]
},
{
  "Effect": "Allow",
  "Action": [
    "lakeformation:GetDataAccess"
  ],
  "Resource": [
    "*
  ]
}
AWS managed policy: AWSQuicksightAthenaAccess

AWSQuicksightAthenaAccess grants access to actions that Amazon QuickSight requires for integration with Athena. You can attach the AWSQuicksightAthenaAccess policy to your IAM identities. Attach this policy only to principals who use Amazon QuickSight with Athena. This policy includes some actions for Athena that are either deprecated and not included in the current public API, or that are used only with the JDBC and ODBC drivers.

Permissions groupings

The AWSQuicksightAthenaAccess policy is grouped into the following sets of permissions.

- **athena** – Allows the principal to run queries on Athena resources.
- **glue** – Allows principals access to AWS Glue databases, tables, and partitions. This is required so that the principal can use the AWS Glue Data Catalog with Athena.
- **s3** – Allows the principal to write and read query results from Amazon S3.
- **lakeformation** – Allows principals to request temporary credentials to access data in a data lake location that is registered with Lake Formation. For more information, see Underlying data access control in the AWS Lake Formation Developer Guide.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "athena:BatchGetQueryExecution",
        "athena:GetQueryExecution",
        "athena:GetQueryResults",
        "athena:GetQueryResultsStream",
        "athena:ListQueryExecutions",
        "athena:StartQueryExecution",
        "athena:StopQueryExecution",
        "athena:ListWorkGroups",
        "athena:ListEngineVersions",
        "athena:GetWorkGroup",
        "athena:GetDataCatalog",
        "athena:GetDatabase",
        "athena:GetTableMetadata",
        "athena:ListDataCatalogs",
        "athena:ListDatabases",
        "athena:ListTableMetadata"
      ],
      "Resource": ["*"
    ],
    {
      "Effect": "Allow",
      "Action": [
        "glue:CreateDatabase",
        "glue:DeleteDatabase",
        "glue:GetDatabase",
        "glue:GetDatabases",
        "glue:UpdateDatabase",
        "glue:GetTable",
        "glue:DeleteTable",
        "glue:BatchDeleteTable",
        "glue:UpdateTable",
        "glue:GetTable"
      ],
      "Resource": ["*"
    ]
  ]
}
```
### Athena updates to AWS managed policies

View details about updates to AWS managed policies for Athena since this service began tracking these changes.

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AmazonAthenaFullAccess (p. 531)</strong>  &lt;br&gt;– Update to existing policy</td>
<td>Athena added <code>cloudwatch:GetMetricData</code> to retrieve CloudWatch metric values. For more information, see <a href="https://docs.aws.amazon.com/AmazonCloudWatch/latest/APIReference/API_GetMetricData.html">GetMetricData</a> in the <a href="https://docs.aws.amazon.com/AmazonCloudWatch/latest/APIReference/">Amazon CloudWatch API Reference</a>.</td>
<td>November 14, 2022</td>
</tr>
<tr>
<td><strong>AmazonAthenaFullAccess (p. 531)</strong>  &lt;br&gt;and  &lt;br&gt;<strong>AWSQuicksightAthenaAccess (p. 534)</strong>  &lt;br&gt;– Updates to existing policies</td>
<td>Athena added <code>s3:PutBucketPublicAccessBlock</code> to enable the blocking of public access.</td>
<td>July 7, 2021</td>
</tr>
</tbody>
</table>
Access through JDBC and ODBC connections

To gain access to AWS services and resources, such as Athena and the Amazon S3 buckets, provide the JDBC or ODBC driver credentials to your application. If you are using the JDBC or ODBC driver, ensure that the IAM permissions policy includes all of the actions listed in AWS managed policy: AWSQuicksightAthenaAccess (p. 534).

Whenever you use IAM policies, make sure that you follow IAM best practices. For more information, see Security best practices in IAM in the IAM User Guide.

Authentication methods

The Athena JDBC and ODBC drivers support SAML 2.0-based authentication, including the following identity providers:

• Active Directory Federation Services (AD FS)
• Azure Active Directory (AD)
• Okta
• PingFederate

For more information, see the installation and configuration guides for the respective drivers, downloadable in PDF format from the JDBC (p. 183) and ODBC (p. 185) driver pages. For additional related information, see the following:

• Enabling federated access to the Athena API (p. 571)
• Using Lake Formation and the Athena JDBC and ODBC drivers for federated access to Athena (p. 589)
• Configuring single sign-on using ODBC, SAML 2.0, and the Okta Identity Provider (p. 204)

For information about the latest versions of the JDBC and ODBC drivers and their documentation, see Using Athena with the JDBC driver (p. 183) and Connecting to Amazon Athena with ODBC (p. 185).

Access to Amazon S3

You can grant access to Amazon S3 locations using identity-based policies, bucket resource policies, access point policies, or any combination of the above.

Whenever you use IAM policies, make sure that you follow IAM best practices. For more information, see Security best practices in IAM in the IAM User Guide.

Note

Athena does not support restricting or allowing access to Amazon S3 resources based on the aws:SourceIp, aws:SourceVpc, or aws:SourceVpce condition keys.

Amazon S3 access points and access point aliases

If you have a shared dataset in an Amazon S3 bucket, maintaining a single bucket policy that manages access for hundreds of use cases can be challenging.
Amazon S3 bucket access points help solve this issue. A bucket can have multiple access points, each with a policy that controls access to the bucket in a different way.

For each access point that you create, Amazon S3 generates an alias that represents the access point. Because the alias is in Amazon S3 bucket name format, you can use the alias in the LOCATION clause of your CREATE TABLE statements in Athena. Athena's access to the bucket is then controlled by the policy for the access point that the alias represents.

For more information, see Table location in Amazon S3 (p. 257) and Using access points in the Amazon S3 User Guide.

**Using CalledVia context keys**

For added security, you can use the aws:CalledVia global condition context key. The aws:CalledVia key contains an ordered list of each service in the chain that made requests on the principal's behalf. By specifying the Athena service principal name athena.amazonaws.com for the aws:CalledVia context key, you can limit requests to only those made from Athena. For more information, see Using Athena with CalledVia context keys (p. 555).

**Additional resources**

For detailed information and examples about how to grant Amazon S3 access, see the following resources:

- Example walkthroughs: Managing access in the Amazon S3 User Guide.
- How can I provide cross-account access to objects that are in Amazon S3 buckets? in the AWS Knowledge Center.
- Cross-account access in Athena to Amazon S3 buckets (p. 537).

**Cross-account access in Athena to Amazon S3 buckets**

A common Amazon Athena scenario is granting access to users in an account different from the bucket owner so that they can perform queries. In this case, use a bucket policy to grant access.

**Note**

For information about cross-account access to AWS Glue data catalogs from Athena, see Cross-account access to AWS Glue data catalogs (p. 548).

The following example bucket policy, created and applied to bucket s3://my-athena-data-bucket by the bucket owner, grants access to all users in account 123456789123, which is a different account.

```json
{
    "Version": "2012-10-17",
    "Id": "MyPolicyID",
    "Statement": [
        {
            "Sid": "MyStatementSid",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::123456789123:root"
            },
            "Action": ["s3:GetBucketLocation", "s3:GetObject", "s3:ListBucket", "s3:ListBucketMultipartUploads"]
        }
    ]
}
```
Cross-account access to S3 buckets

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ExamplePolicy1",
      "Action": [
        "s3:ListMultipartUploadParts",
        "s3:AbortMultipartUpload",
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::my-athena-data-bucket",
        "arn:aws:s3:::my-athena-data-bucket/*"
      ]
    }
  ]
}
```

To grant access to a particular user in an account, replace the Principal key with a key that specifies the user instead of root. For example, for user profile Dave, use arn:aws:iam::123456789123:user/Dave.

**Cross-account access to a bucket encrypted with a custom AWS KMS key**

If you have an Amazon S3 bucket that is encrypted with a custom AWS Key Management Service (AWS KMS) key, you might need to grant access to it to users from another Amazon Web Services account.

Granting access to an AWS KMS-encrypted bucket in Account A to a user in Account B requires the following permissions:

- The bucket policy in Account A must grant access to Account B.
- The AWS KMS key policy in Account A must grant access to the user in Account B.
- The AWS Identity and Access Management (IAM) user policy in Account B must grant the user access to both the bucket and the key in Account A.

The following procedures describe how to grant each of these permissions.

**To grant access to the bucket in account a to the user in account b**

- From Account A, review the S3 bucket policy and confirm that there is a statement that allows access from the account ID of Account B.

For example, the following bucket policy allows s3:GetObject access to the account ID 111122223333:

```
{
  "Id": "ExamplePolicy1",
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ExampleStmt1",
      "Action": ["s3:GetObject"],
      "Effect": "Allow",
      "Resource": "arn:aws:s3:::awsexamplebucket/*",
      "Principal": {
        "AWS": ["111122223333"]
      }
    }
  ]
}
```
To grant access to the user in account b from the AWS KMS key policy in account a

1. In the AWS KMS key policy for Account A, grant the user in Account B permissions to the following actions:
   - kms:Encrypt
   - kms:Decrypt
   - kms:ReEncrypt*
   - kms:GenerateDataKey*
   - kms:DescribeKey

The following example grants key access to only one IAM user or role.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "AllowUseOfTheKey",
         "Effect": "Allow",
         "Principal": {
            "AWS": "arn:aws:iam::111122223333:role/role_name"
         },
         "Action": [
            "kms:Encrypt",
            "kms:Decrypt",
            "kms:ReEncrypt*",
            "kms:GenerateDataKey*",
            "kms:DescribeKey"
         ],
         "Resource": "*"
      }
   ],
   "Resource": "*"
}
```

2. From Account A, review the key policy using the AWS Management Console policy view.

3. In the key policy, verify that the following statement lists Account B as a principal.
   
   "Sid": "Allow use of the key"

4. If the "Sid": "Allow use of the key" statement is not present, perform the following steps:
   a. Switch to view the key policy using the console default view.
   b. Add Account B's account ID as an external account with access to the key.

To grant access to the bucket and the key in account a from the IAM user policy in account b

1. From Account B, open the IAM console at https://console.aws.amazon.com/iam/.

2. Open the IAM user or role associated with the user in Account B.

3. Review the list of permissions policies applied to IAM user or role.

4. Ensure that a policy is applied that grants access to the bucket.

The following example statement grants the IAM user access to the s3:GetObject and s3:PutObject operations on the bucket awsexamplebucket:

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "AllowS3Access",
         "Effect": "Allow",
         "Principal": {
            "AWS": "arn:aws:iam::111122223333:user/234567890123:alias/iam_user"
         },
         "Action": [
            "s3:GetObject",
            "s3:PutObject"
         ],
         "Resource": "arn:aws:s3:::awsexamplebucket/*"
      }
   ],
   "Resource": "*"
}
```
5. Ensure that a policy is applied that grants access to the key.

**Note**  
If the IAM user or role in Account B already has administrator access, then you don't need to grant access to the key from the user's IAM policies.

The following example statement grants the IAM user access to use the key `arn:aws:kms:us-west-2:123456789098:key/111aa2bb-333c-4d44-5555-a111bb2c33dd`.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "ExampleStmt3",
         "Action": [
            "kms:Decrypt",
            "kms:DescribeKey",
            "kms:Encrypt",
            "kms:GenerateDataKey",
            "kms:ReEncrypt*"
         ],
         "Effect": "Allow",
         "Resource": "arn:aws:kms:us-west-2:123456789098:key/111aa2bb-333c-4d44-5555-a111bb2c33dd"
      }
   }
}
```

For instructions on how to add or correct the IAM user's permissions, see Changing permissions for an IAM user.

**Cross-account access to bucket objects**

Objects that are uploaded by an account (Account C) other than the bucket's owning account (Account A) might require explicit object-level ACLs that grant read access to the querying account (Account B). To avoid this requirement, Account C should assume a role in Account A before it places objects in Account A's bucket. For more information, see How can I provide cross-account access to objects that are in Amazon S3 buckets?

**Fine-grained access to databases and tables in the AWS Glue Data Catalog**

If you use the AWS Glue Data Catalog with Amazon Athena, you can define resource-level policies for the database and table Data Catalog objects that are used in Athena.
**Note**
The term "fine-grained access control" here refers to database and table level security. For information about column-, row-, and cell-level security, see Data filtering and cell-level security in Lake Formation.

You define resource-level permissions in IAM identity-based policies.

**Important**
This section discusses resource-level permissions in IAM identity-based policies. These are different from resource-based policies. For more information about the differences, see Identity-based policies and resource-based policies in the IAM User Guide.

See the following topics for these tasks:

<table>
<thead>
<tr>
<th>To perform this task</th>
<th>See the following topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create an IAM policy that defines fine-grained access to resources</td>
<td>Creating IAM policies in the IAM User Guide.</td>
</tr>
<tr>
<td>Learn about IAM identity-based policies used in AWS Glue</td>
<td>Identity-based policies (IAM policies) in the AWS Glue Developer Guide.</td>
</tr>
</tbody>
</table>

**In this section**

- Limitations (p. 541)
- Mandatory: Access policy to the Default database and catalog per AWS Region (p. 542)
- Table partitions and versions in AWS Glue (p. 543)
- Examples of fine-grained permissions to tables and databases (p. 543)

**Limitations**

Consider the following limitations when using fine-grained access control with the AWS Glue Data Catalog and Athena:

- You can limit access only to databases and tables. Fine-grained access controls apply at the table level and you cannot limit access to individual partitions within a table. For more information, see Table partitions and versions in AWS Glue (p. 543).
- The AWS Glue Data Catalog contains the following resources: CATALOG, DATABASE, TABLE, and FUNCTION.

**Note**
From this list, resources that are common between Athena and the AWS Glue Data Catalog are TABLE, DATABASE, and CATALOG for each account. Function is specific to AWS Glue. For delete actions in Athena, you must include permissions to AWS Glue actions. See Examples of fine-grained permissions to tables and databases (p. 543).

The hierarchy is as follows: CATALOG is an ancestor of all DATABASES in each account, and each DATABASE is an ancestor for all of its TABLES and FUNCTIONS. For example, for a table named table_test that belongs to a database db in the catalog in your account, its ancestors are db and the catalog in your account. For the db database, its ancestor is the catalog in your account, and its descendants are tables and functions. For more information about the hierarchical structure of resources, see List of ARNs in Data Catalog in the AWS Glue Developer Guide.
For any non-delete Athena action on a resource, such as CREATE DATABASE, CREATE TABLE, SHOW DATABASE, SHOW TABLE, or ALTER TABLE, you need permissions to call this action on the resource (table or database) and all ancestors of the resource in the Data Catalog. For example, for a table, its ancestors are the database to which it belongs, and the catalog for the account. For a database, its ancestor is the catalog for the account. See Examples of fine-grained permissions to tables and databases (p. 543).

For a delete action in Athena, such as DROP DATABASE or DROP TABLE, you also need permissions to call the delete action on all ancestors and descendants of the resource in the Data Catalog. For example, to delete a database you need permissions on the database, the catalog, which is its ancestor, and all the tables and user defined functions, which are its descendents. A table does not have descendents. To run DROP TABLE, you need permissions to this action on the table, the database to which it belongs, and the catalog. See Examples of fine-grained permissions to tables and databases (p. 543).

When limiting access to a specific database in the Data Catalog, you must also specify the access policy to the default database and catalog for each AWS Region for GetDatabase and CreateDatabase actions. If you use Athena in more than one Region, add a separate line to the policy for the resource ARN for each default database and catalog in each Region.

For example, to allow GetDatabase access to example_db in the us-east-1 (N.Virginia) Region, also include the default database and catalog in the policy for that Region for the actions GetDatabase, GetDatabases, and CreateDatabase:

```json
{
   "Effect": "Allow",
   "Action": [
      "glue:GetDatabase",
      "glue:GetDatabases",
      "glue:CreateDatabase"
   ],
   "Resource": [
      "arn:aws:glue:us-east-1:123456789012:catalog",
      "arn:aws:glue:us-east-1:123456789012:database/example_db"
   ]
}
```

**Mandatory: Access policy to the Default database and catalog per AWS Region**

For Athena to work with the AWS Glue Data Catalog, the following access policy to the default database and to the AWS Glue Data Catalog per AWS Region for GetDatabase, GetDatabases, and CreateDatabase must be present:

```json
{
   "Effect": "Allow",
   "Action": [
      "glue:GetDatabase",
      "glue:GetDatabases",
      "glue:CreateDatabase"
   ],
   "Resource": [
      "arn:aws:glue:us-east-1:123456789012:catalog",
      "arn:aws:glue:us-east-1:123456789012:database/default"
   ]
}
```
Table partitions and versions in AWS Glue

In AWS Glue, tables can have partitions and versions. Table versions and partitions are not considered to be independent resources in AWS Glue. Access to table versions and partitions is given by granting access on the table and ancestor resources for the table.

For the purposes of fine-grained access control, the following access permissions apply:

- Fine-grained access controls apply at the table level. You can limit access only to databases and tables. For example, if you allow access to a partitioned table, this access applies to all partitions in the table. You cannot limit access to individual partitions within a table.

  **Important**
  Having access to all partitions within a table is not sufficient if you need to run actions in AWS Glue on partitions. To run actions on partitions, you need permissions for those actions. For example, to run GetPartitions on table myTable in the database myDB, you need permissions for the action `glue:GetPartitions` in the Data Catalog, the myDB database, and myTable.

- Fine-grained access controls do not apply to table versions. As with partitions, access to previous versions of a table is granted through access to the table version APIs in AWS Glue on the table, and to the table ancestors.

For information about permissions on AWS Glue actions, see AWS Glue API permissions: Actions and resources reference in the AWS Glue Developer Guide.

Examples of fine-grained permissions to tables and databases

The following table lists examples of IAM identity-based policies that allow fine-grained access to databases and tables in Athena. We recommend that you start with these examples and, depending on your needs, adjust them to allow or deny specific actions to particular databases and tables.

These examples include the access policy to the default database and catalog, for GetDatabase and CreateDatabase actions. This policy is required for Athena and the AWS Glue Data Catalog to work together. For multiple AWS Regions, include this policy for each of the default databases and their catalogs, one line for each Region.

In addition, replace the example_db database and test table names with the names for your databases and tables.

<table>
<thead>
<tr>
<th>DDL statement</th>
<th>Example of an IAM access policy granting access to the resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER DATABASE</td>
<td>Allows you to modify the properties for the example_db database.</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>&quot;Effect&quot;: &quot;Allow&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;Action&quot;: [</td>
</tr>
<tr>
<td></td>
<td>&quot;glue:GetDatabase&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;glue:CreateDatabase&quot;</td>
</tr>
<tr>
<td></td>
<td>],</td>
</tr>
<tr>
<td></td>
<td>&quot;Resource&quot;: [</td>
</tr>
<tr>
<td></td>
<td>&quot;arn:aws:glue:us-east-1:123456789012:catalog&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;arn:aws:glue:us-east-1:123456789012:database/default&quot;</td>
</tr>
<tr>
<td></td>
<td>]</td>
</tr>
<tr>
<td></td>
<td>},</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>&quot;Effect&quot;: &quot;Allow&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;Action&quot;: [</td>
</tr>
<tr>
<td></td>
<td>&quot;glue:GetDatabase&quot;,</td>
</tr>
<tr>
<td></td>
<td>]</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>DDL statement</td>
<td>Example of an IAM access policy granting access to the resource</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------------------------------------------------</td>
</tr>
<tr>
<td>CREATE DATABASE</td>
<td>Allows you to create the database named example_db.</td>
</tr>
<tr>
<td>CREATE TABLE</td>
<td>Allows you to create a table named test in the example_db database.</td>
</tr>
</tbody>
</table>
### DDL statement

Table: Example of an IAM access policy granting access to the resource

<table>
<thead>
<tr>
<th>DDL statement</th>
<th>Example of an IAM access policy granting access to the resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>DROP DATABASE</td>
<td>Allows you to drop the example_db database, including all tables in it.</td>
</tr>
</tbody>
</table>

```json
{
    "Effect": "Allow",
    "Action": [
        "glue:GetDatabase",
        "glue:CreateDatabase"
    ],
    "Resource": [
        "arn:aws:glue:us-east-1:123456789012:catalog",
        "arn:aws:glue:us-east-1:123456789012:database/default"
    ],
},
{
    "Effect": "Allow",
    "Action": [
        "glue:GetDatabase",
        "glue:DeleteDatabase",
        "glue:GetTables",
        "glue:GetTable",
        "glue:DeleteTable"
    ],
    "Resource": [
        "arn:aws:glue:us-east-1:123456789012:catalog",
        "arn:aws:glue:us-east-1:123456789012:database/example_db",
        "arn:aws:glue:us-east-1:123456789012:table/example_db/*",
        "arn:aws:glue:us-east-1:123456789012:userDefinedFunction/example_db/*"
    ]
}
```
<table>
<thead>
<tr>
<th>DDL statement</th>
<th>Example of an IAM access policy granting access to the resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>DROP TABLE</td>
<td>Allows you to drop a partitioned table named test in the example_db database. If your table does not have partitions, do not include partition actions.</td>
</tr>
</tbody>
</table>

```json
{
  "Effect": "Allow",
  "Action": [
    "glue:GetDatabase",
    "glue:CreateDatabase"
  ],
  "Resource": [
    "arn:aws:glue:us-east-1:123456789012:catalog",
    "arn:aws:glue:us-east-1:123456789012:database/default"
  ]
},
{
  "Effect": "Allow",
  "Action": [
    "glue:GetDatabase",
    "glue:GetTable",
    "glue:DeleteTable",
    "glue:GetPartitions",
    "glue:GetPartition",
    "glue:DeletePartition"
  ],
  "Resource": [
    "arn:aws:glue:us-east-1:123456789012:catalog",
    "arn:aws:glue:us-east-1:123456789012:database/example_db",
    "arn:aws:glue:us-east-1:123456789012:table/example_db/test"
  ]
}
```
<table>
<thead>
<tr>
<th>DDL statement</th>
<th>Example of an IAM access policy granting access to the resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSCK REPAIR TABLE</td>
<td>Allows you to update catalog metadata after you add Hive compatible partitions to the table named test in the example_db database.</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>&quot;Effect&quot;: &quot;Allow&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;Action&quot;: [</td>
</tr>
</tbody>
</table>
|                       |     "glue:GetDatabase",
|                       |     "glue:CreateDatabase"
|                       |   ],                                                                                                                         |
|                       |   "Resource": [                                                                                                                |
|                       |     "arn:aws:glue:us-east-1:123456789012:catalog",
|                       |     "arn:aws:glue:us-east-1:123456789012:database/default"                                                                 |
|                       | ]                                                                                                                                 |
|                       | },                                                                                                                           |
|                       |   {                                                                                                                           |
|                       |     "Effect": "Allow",
|                       |     "Action": [                                                                                                              |
|                       |       "glue:GetDatabase",
|                       |       "glue:GetTable",
|                       |       "glue:GetPartitions",
|                       |       "glue:GetPartition",
|                       |       "glue:BatchCreatePartition"
|                       |   ],                                                                                                                         |
|                       |   "Resource": [                                                                                                                |
|                       |     "arn:aws:glue:us-east-1:123456789012:catalog",
|                       |     "arn:aws:glue:us-east-1:123456789012:database/example_db",
|                       |     "arn:aws:glue:us-east-1:123456789012:table/example_db/test"                                                                |
|                       | ]                                                                                                                           |
| SHOW DATABASES        | Allows you to list all databases in the AWS Glue Data Catalog.                                                                  |
|                       | {                                                                                                                                 |
|                       |   "Effect": "Allow",
|                       |   "Action": [                                                                                                              |
|                       |     "glue:GetDatabase",
|                       |     "glue:CreateDatabase"
|                       |   ],                                                                                                                         |
|                       |   "Resource": [                                                                                                                |
|                       |     "arn:aws:glue:us-east-1:123456789012:catalog",
|                       |     "arn:aws:glue:us-east-1:123456789012:database/default"                                                                 |
|                       | ]                                                                                                                                 |
|                       | },                                                                                                                           |
|                       |   {                                                                                                                           |
|                       |     "Effect": "Allow",
|                       |     "Action": [                                                                                                              |
|                       |       "glue:GetDatabases"
|                       |   ],                                                                                                                         |
|                       |   "Resource": [                                                                                                                |
|                       |     "arn:aws:glue:us-east-1:123456789012:catalog",
|                       |     "arn:aws:glue:us-east-1:123456789012:database/*"                                                                            |
|                       | ]                                                                                                                           |
|                       | }                                                                                                                                 |
### Cross-account access to AWS Glue data catalogs

You can use Athena’s cross-account AWS Glue catalog feature to register an AWS Glue catalog from an account other than your own. After you configure the required IAM permissions for AWS Glue and register the catalog as an Athena `DataCatalog` resource, you can use Athena to run cross-account queries. For information about using the Athena console to register a catalog from another account, see Registering an AWS Glue Data Catalog from another account (p. 24).

For more information about cross-account access in AWS Glue, see Granting cross-account access in the AWS Glue Developer Guide.

### Before you start

Because this feature uses existing Athena `DataCatalog` resource APIs and functionality to enable cross-account access, we recommend that you read the following resources before you start:

- Connecting to data sources (p. 21) - Contains topics on using Athena with AWS Glue, Hive, or Lambda data catalog sources.
- Data Catalog example policies (p. 179) - Shows how to write policies that control access to data catalogs.
- Using the AWS CLI with Hive metastores (p. 54) - Shows how to use the AWS CLI with Hive metastores, but contains use cases applicable to other data sources.

### Considerations and limitations

Currently, Athena cross-account AWS Glue catalog access has the following limitations:

<table>
<thead>
<tr>
<th>DDL statement</th>
<th>Example of an IAM access policy granting access to the resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHOW TABLES</td>
<td>Allows you to list all tables in the <code>example_db</code> database.</td>
</tr>
<tr>
<td></td>
<td>`{</td>
</tr>
<tr>
<td></td>
<td>&quot;Effect&quot;: &quot;Allow&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;Action&quot;: [</td>
</tr>
<tr>
<td></td>
<td>&quot;glue:GetDatabase&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;glue:CreateDatabase&quot;</td>
</tr>
<tr>
<td></td>
<td>},</td>
</tr>
<tr>
<td></td>
<td>&quot;Resource&quot;: [</td>
</tr>
<tr>
<td></td>
<td>&quot;arn:aws:glue:us-east-1:123456789012:catalog&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;arn:aws:glue:us-east-1:123456789012:database/default&quot;</td>
</tr>
<tr>
<td></td>
<td>]</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>`{</td>
</tr>
<tr>
<td></td>
<td>&quot;Effect&quot;: &quot;Allow&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;Action&quot;: [</td>
</tr>
<tr>
<td></td>
<td>&quot;glue:GetDatabase&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;glue:GetTables&quot;</td>
</tr>
<tr>
<td></td>
<td>},</td>
</tr>
<tr>
<td></td>
<td>&quot;Resource&quot;: [</td>
</tr>
<tr>
<td></td>
<td>&quot;arn:aws:glue:us-east-1:123456789012:catalog&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;arn:aws:glue:us-east-1:123456789012:database/example_db&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;arn:aws:glue:us-east-1:123456789012:table/example_db/*&quot;</td>
</tr>
<tr>
<td></td>
<td>]</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
• The feature is available only in AWS Regions where Athena engine version 2 or later is supported. For information about Athena engine versions, see Athena engine versioning (p. 669). To upgrade the engine version for a workgroup, see Changing Athena engine versions (p. 669).

• When you register another account's AWS Glue Data Catalog in your account, you create a regional DataCatalog resource that is linked to the other account's data in that particular Region only.

• Currently, CREATE VIEW statements that include a cross-account AWS Glue catalog are not supported.

**Getting started**

In the following scenario, the "Borrower" account (666666666666) wants to run a SELECT query that refers to the AWS Glue catalog that belongs to the "Owner" account (999999999999), as in the following example:

```
SELECT * FROM ownerCatalog.tpch1000.customer
```

In the following procedure, Steps 1a and 1b show how to give the Borrower account access to the Owner account's AWS Glue resources, from both the Owner's perspective and from the Borrower's perspective. The example grants access to the database tpch1000 and the table customer. Change these example names to fit your requirements.

**Step 1a: Create an owner policy to grant AWS Glue access to the borrower**

To grant AWS Glue access from the Owner account (999999999999) to the Borrower's user or role, you can use the AWS Glue console or the AWS Glue PutResourcePolicy API operation. The following procedure uses the AWS Glue console.

**To grant AWS Glue access to the borrower account from the owner**

1. Sign in to the AWS Glue console at https://console.aws.amazon.com/glue/ from the Owner account.
2. In the navigation pane, choose Settings.
3. In the Permissions box, enter a policy like the following. If you want to increase the permission scope, you can use the wildcard character * for both the database and table resource types.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "arn:aws:iam::666666666666:user/username",
                    "arn:aws:iam::666666666666:role/rolename"
                ]
            },
            "Action": "glue:*",
            "Resource": [
                "arn:aws:glue:us-east-1:999999999999:catalog",
                "arn:aws:glue:us-east-1:999999999999:table/tpch1000/customer"
            ]
        }
    ]
}
```
Step 1b: Create a borrower role or user policy with access to the owner’s AWS Glue resources

To give the Borrower account role or user access to the Owner account’s AWS Glue resources, you can use the AWS Identity and Access Management (IAM) console or the IAM API. The following procedures use the IAM console.

To grant a borrower role or user access to the owner account’s AWS Glue resources

2. In the navigation pane, choose Roles or Users.
3. Choose the role or user that you want to change.
4. Do one of the following:
   • If you are changing a role, choose Attach policies.
   • If you are changing a user, choose Add permissions, and then choose Attach existing policies directly.
5. Choose Create policy.
6. Choose JSON.
7. In the box, enter the following policy, and then modify it according to your requirements:

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": "glue:*",
         "Resource": [
            "arn:aws:glue:us-east-1:999999999999:catalog",
            "arn:aws:glue:us-east-1:999999999999:table/tpch1000/customer"
         ]
      }
   ]
}
```
9. For Name, enter a name for the policy.
10. Choose Create policy.

After you finish, it is recommend that you use the AWS Glue API to make some test cross-account calls to confirm that permissions are configured as you expect.

Step 2: The borrower registers the AWS Glue Data Catalog that belongs to the owner account

The following procedure shows you how to use the Athena console to configure the AWS Glue Data Catalog in the owner Amazon Web Services account as a data source. For information about using API operations instead of the console to register the catalog, see Using the API to register an Athena Data Catalog that belongs to the owner account (p. 552).

To register an AWS Glue Data Catalog belonging to another account

2. If the console navigation pane is not visible, choose the expansion menu on the left.
3. Choose **Data sources**.
4. On the upper right, choose **Create data source**.
5. On the **Choose a data source** page, for **Data sources**, choose **S3 - AWS Glue Data Catalog**, and then choose **Next**.
6. On the **Enter data source details** page, in the **AWS Glue Data Catalog** section, for **Choose an AWS Glue Data Catalog**, choose **AWS Glue Data Catalog in another account**.
7. For **Data source details**, enter the following information:
   - **Data source name** – Enter the name that you want to use in your SQL queries to refer to the data catalog in the other account.
   - **Description** – (Optional) Enter a description of the data catalog in the other account.
   - **Catalog ID** – Enter the 12-digit Amazon Web Services account ID of the account to which the data catalog belongs. The Amazon Web Services account ID is the catalog ID.
8. (Optional) For **Tags**, enter key-value pairs that you want to associate with the data source. For more information about tags, see **Tagging Athena resources** (p. 658).
9. Choose **Next**.
10. On the **Review and create** page, review the information that you provided, and then choose **Create data source**. The **Data source details** page lists the databases and tags for the data catalog that you registered.
11. Choose **Data sources**. The data catalog that you registered is listed in the **Data source name** column.
12. To view or edit information about the data catalog, choose the catalog, and then choose **Actions, Edit**.
13. To delete the new data catalog, choose the catalog, and then choose **Actions, Delete**.

**Step 3: The borrower submits a query**

The Borrower submits a query that references the catalog using the `catalog.database.table` syntax, as in the following example:

```
SELECT * FROM ownerCatalog.tpch1000.customer
```

Instead of using the fully qualified syntax, the Borrower can also specify the catalog contextually by passing it in through the `QueryExecutionContext`.

**Additional Amazon S3 permissions**

- If the Borrower account uses an Athena query to write new data to a table in the Owner account, the Owner will not automatically have access to this data in Amazon S3, even though the table exists in the Owner’s account. This is because the Borrower is the object owner of the information in Amazon S3.


S3 unless otherwise configured. To grant the Owner access to the data, set the permissions on the objects accordingly as an additional step.

- Certain cross-account DDL operations like MSCK REPAIR TABLE (p. 756) require Amazon S3 permissions. For example, if the Borrower account is performing a cross-account MSCK REPAIR operation against a table in the Owner account that has its data in an Owner account S3 bucket, that bucket must grant permissions to the Borrower for the query to succeed.

For information about granting bucket permissions, see How do I set ACL bucket permissions? in the Amazon Simple Storage Service User Guide.

Using a catalog dynamically

In some cases you might want to quickly perform testing against a cross-account AWS Glue catalog without the prerequisite step of registering it. You can dynamically perform cross-account queries without creating the DataCatalog resource object if the required IAM and Amazon S3 permissions are correctly configured as described earlier in this document.

To explicitly reference a catalog without registration, use the syntax in the following example:

```sql
SELECT * FROM "glue:arn:aws:glue:us-east-1:999999999999:catalog".tpch1000.customer
```

Use the format "glue:<arn>“, where <arn> is the AWS Glue Data Catalog ARN that you want to use. In the example, Athena uses this syntax to dynamically point to account 999999999999's AWS Glue data catalog as if you had separately created a DataCatalog object for it.

Notes for using dynamic catalogs

When you use dynamic catalogs, remember the following points.

- Use of a dynamic catalog requires the IAM permissions that you normally use for Athena Data Catalog API operations. The main difference is that the Data Catalog resource name follows the glue:* naming convention.
- The catalog ARN must belong to the same Region where the query is being run.
- When using a dynamic catalog in a DML query or view, surround it with escaped double quotation marks ("\"). When using a dynamic catalog in a DDL query, surround it with backtick characters (‘`).

Using the API to register an Athena Data Catalog that belongs to the owner account

Instead of using the Athena console as described in Step 2, it is possible to use API operations to register the Data Catalog that belongs to the owner account.

The creator of the Athena DataCatalog resource must have the necessary permissions to run the Athena CreateDataCatalog API operation. Depending on your requirements, access to additional API operations might be necessary. For more information, see Data Catalog example policies (p. 179).

The following CreateDataCatalog request body registers an AWS Glue catalog for cross-account access:

```json
# Example CreateDataCatalog request to register a cross-account Glue catalog:
{
    "Description": "Cross-account Glue catalog",
    "Name": "ownerCatalog",
    "Parameters": {
        "catalog-id": "999999999999"  # Owner's account ID
    },
    "Type": "GLUE"
}
```
The following sample code uses a Java client to create the DataCatalog object.

```java
# Sample code to create the DataCatalog through Java client
CreateDataCatalogRequest request = new CreateDataCatalogRequest()
    .withName("ownerCatalog")
    .withType(DataCatalogType.GLUE)
    .withParameters(ImmutableMap.of("catalog-id", "999999999999"));

athenaClient.createDataCatalog(request);
```

After these steps, the borrower should see ownerCatalog when it calls the ListDataCatalogs API operation.

**Access from Athena to encrypted metadata in the AWS Glue Data Catalog**

If you use the AWS Glue Data Catalog with Amazon Athena, you can enable encryption in the AWS Glue Data Catalog using the AWS Glue console or the API. For information, see Encrypting your data catalog in the AWS Glue Developer Guide.

If the AWS Glue Data Catalog is encrypted, you must add the following actions to all policies that are used to access Athena:

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "kms:GenerateDataKey",
            "kms:Decrypt",
            "kms:Encrypt"
         ],
         "Resource": "(arn of the key used to encrypt the catalog)"
      }
   ]
}
```

Whenever you use IAM policies, make sure that you follow IAM best practices. For more information, see Security best practices in IAM in the IAM User Guide.

**Access to workgroups and tags**

A workgroup is a resource managed by Athena. Therefore, if your workgroup policy uses actions that take workgroup as an input, you must specify the workgroup's ARN as follows, where `workgroup-name` is the name of your workgroup:

```
```

For example, for a workgroup named `test_workgroup` in the `us-west-2` region for Amazon Web Services account `123456789012`, specify the workgroup as a resource using the following ARN:

```
```

- For a list of workgroup policies, see the section called “Workgroup example policies” (p. 633).
- For a list of tag-based policies for workgroups, see Tag-based IAM access control policies (p. 664).
Allow access to prepared statements

This topic covers IAM permissions for prepared statements in Amazon Athena. Whenever you use IAM policies, make sure that you follow IAM best practices. For more information, see Security best practices in IAM in the IAM User Guide.

The following IAM permissions are required for creating, managing, and executing prepared statements.

<table>
<thead>
<tr>
<th>To do this</th>
<th>Use these permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run a PREPARE query</td>
<td>athena:StartQueryExecution</td>
</tr>
<tr>
<td></td>
<td>athena:CreatePreparedStatement</td>
</tr>
<tr>
<td>Re-run a PREPARE query to update an existing prepared statement</td>
<td>athena:StartQueryExecution</td>
</tr>
<tr>
<td></td>
<td>athena:UpdatePreparedStatement</td>
</tr>
<tr>
<td>Run an EXECUTE query</td>
<td>athena:StartQueryExecution</td>
</tr>
<tr>
<td></td>
<td>athena:GetPreparedStatement</td>
</tr>
<tr>
<td>Run a DEALLOCATE PREPARE query</td>
<td>athena:StartQueryExecution</td>
</tr>
<tr>
<td></td>
<td>athena:DeletePreparedStatement</td>
</tr>
</tbody>
</table>

Example

The following example IAM policy grants permissions to manage and run prepared statements on a specified account ID and workgroup.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "athena:StartQueryExecution",
                "athena:CreatePreparedStatement",
                "athena:UpdatePreparedStatement"
            ]
        }
    ]
}
```
Using Athena with CalledVia context keys

When a principal makes a request to AWS, AWS gathers the request information into a request context that evaluates and authorizes the request. You can use the Condition element of a JSON policy to compare keys in the request context with key values that you specify in your policy. Global condition context keys are condition keys with an aws: prefix.

The aws:CalledVia context key

You can use the aws:CalledVia global condition context key to compare the services in the policy with the services that made requests on behalf of the IAM principal (user or role). When a principal makes a request to an AWS service, that service might use the principal's credentials to make subsequent requests to other services. The aws:CalledVia key contains an ordered list of each service in the chain that made requests on the principal's behalf.

By specifying a service principal name for the aws:CalledVia context key, you can make the context key AWS service-specific. For example, you can use the aws:CalledVia condition key to limit requests to only those made from Athena. To use the aws:CalledVia condition key in a policy with Athena, you specify the Athena service principal name athena.amazonaws.com, as in the following example.

```json
...  "Condition": {    "ForAnyValue:StringEquals": {      "aws:CalledVia": "athena.amazonaws.com"    }  } ...
```

You can use the aws:CalledVia context key to ensure that callers only have access to a resource (like a Lambda function) if they call the resource from Athena.

Add an optional CalledVia context key for fine grained access to a Lambda function

Athena requires the caller to have lambda:InvokeFunction permissions in order to invoke the Lambda function associated with the query. The following statement allows fine-grained access to a Lambda function so that the user can use only Athena to invoke the Lambda function.

```json
```
Using CalledVia context keys

```

"Condition": {
    "ForAnyValue:StringEquals": {
        "aws:CalledVia": "athena.amazonaws.com"
    }
  }
}
```

The following example shows the addition of the previous statement to a policy that allows a user to run and read a federated query. Principals who are allowed to perform these actions can run queries that specify Athena catalogs associated with a federated data source. However, the principal cannot access the associated Lambda function unless the function is invoked through Athena.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": [
                "athena:GetWorkGroup",
                "s3:PutObject",
                "s3:GetObject",
                "athena:StartQueryExecution",
                "s3:AbortMultipartUpload",
                "athena:StopQueryExecution",
                "athena:GetQueryExecution",
                "athena:GetQueryResults",
                "s3:ListMultipartUploadParts"
            ],
            "Resource": [
                "arn:aws:athena:*:111122223333:workgroup/WorkGroupName",
                "arn:aws:s3:::MyQueryResultsBucket/*",
                "arn:aws:s3:::MyLambdaSpillBucket/MyLambdaSpillPrefix/*"
            ]
        },
        {
            "Sid": "VisualEditor1",
            "Effect": "Allow",
            "Action": "athena:ListWorkGroups",
            "Resource": "*"
        },
        {
            "Sid": "VisualEditor2",
            "Effect": "Allow",
            "Action": [
                "s3:ListBucket",
                "s3:GetBucketLocation"
            ],
            "Resource": "arn:aws:s3:::MyLambdaSpillBucket"
        },
        {
            "Sid": "VisualEditor3",
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": [
                "arn:aws:lambda:*:111122223333:function:OneAthenaLambdaFunction",
                "arn:aws:lambda:*:111122223333:function:AnotherAthenaLambdaFunction"
            ],
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "aws:CalledVia": "athena.amazonaws.com"
                }
            }
        }
    ]
}
```
For more information about CalledVia condition keys, see AWS global condition context keys in the IAM User Guide.

Allow access to an Athena Data Connector for External Hive Metastore

The permission policy examples in this topic demonstrate required allowed actions and the resources for which they are allowed. Examine these policies carefully and modify them according to your requirements before you attach similar permissions policies to IAM identities.

- Example Policy to Allow an IAM Principal to Query Data Using Athena Data Connector for External Hive Metastore (p. 557)
- Example Policy to Allow an IAM Principal to Create an Athena Data Connector for External Hive Metastore (p. 558)

Example – Allow an IAM principal to query data using Athena Data Connector for External Hive Metastore

The following policy is attached to IAM principals in addition to the AWS managed policy: AmazonAthenaFullAccess (p. 531), which grants full access to Athena actions.

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "VisualEditor1",
         "Effect": "Allow",
         "Action": [
            "lambda:GetFunction",
            "lambda:GetLayerVersion",
            "lambda:InvokeFunction"
         ],
         "Resource": [
            "arn:aws:lambda:*:111122223333:function:MyAthenaLambdaFunction",
            "arn:aws:lambda:*:111122223333:function:AnotherAthenaLambdaFunction",
            "arn:aws:lambda:*:111122223333:layer:MyAthenaLambdaLayer:*"
         ]
      },
      {
         "Sid": "VisualEditor2",
         "Effect": "Allow",
         "Action": [
            "s3:GetBucketLocation",
            "s3:GetObject",
            "s3:ListBucket",
            "s3:PutObject",
            "s3:ListMultipartUploadParts",
            "s3:AbortMultipartUpload"
         ],
         "Resource": "arn:aws:s3:::MyLambdaSpillBucket/MyLambdaSpillLocation"
      }
   ]
}
```
Explanation of permissions

<table>
<thead>
<tr>
<th>Allowed actions</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;s3:GetBucketLocation&quot;,</td>
<td>s3 actions allow reading from and</td>
</tr>
<tr>
<td>&quot;s3:GetObject&quot;,</td>
<td>writing to the resource specified as</td>
</tr>
<tr>
<td>&quot;s3:ListBucket&quot;,</td>
<td>&quot;arn:aws:s3:::MyLambdaSpillBucket/MyLambdaSpillLocation&quot;, where</td>
</tr>
<tr>
<td>&quot;s3:PutObject&quot;,</td>
<td>MyLambdaSpillLocation identifies the spill bucket that is specified in the</td>
</tr>
<tr>
<td>&quot;s3:ListMultipartUploadParts&quot;,</td>
<td>configuration of the Lambda function or functions being invoked. The</td>
</tr>
<tr>
<td>&quot;s3:AbortMultipartUpload&quot;</td>
<td>resource identifier is required only if you use a Lambda layer to create</td>
</tr>
<tr>
<td></td>
<td>custom runtime dependencies to reduce function artifact size at deploy</td>
</tr>
<tr>
<td></td>
<td>ment time. The * in the last position is a wildcard for layer version.</td>
</tr>
</tbody>
</table>

| "lambda:GetFunction",                | Allows queries to invoke the AWS                                            |
| "lambda:GetLayerVersion",            | Lambda functions specified in the                                          |
| "lambda:InvokeFunction"              | Resource block. For example,                                               |
|                                      | arn:aws:lambda:*:MyAWSAcctId:function:MyAthenaLambdaFunction,              |
|                                      | where MyAthenaLambdaFunction specifies the name of a Lambda function to be  |
|                                      | invoked. Multiple functions can be specified as shown in the example.       |

Example – Allow an IAM principal to create an Athena Data Connector for External Hive Metastore

The following policy is attached to IAM principals in addition to the AWS managed policy: AmazonAthenaFullAccess (p. 531), which grants full access to Athena actions.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "lambda:GetFunction",
        "lambda:ListFunctions",
        "lambda:GetLayerVersion",
        "lambda:InvokeFunction",
        "lambda:CreateFunction",
        "lambda:DeleteFunction",
        "lambda:PublishLayerVersion",
        "lambda:DeleteLayerVersion",
        "lambda:UpdateFunctionConfiguration",
        "lambda:PutFunctionConcurrency",
        "lambda:DeleteFunctionConcurrency"
      ],
      "Resource": "arn:aws:lambda:*:111122223333:function:MyAthenaLambdaFunctionsPrefix"
    }
  ]
}
```

Explanation of Permissions
Allow Lambda function access to external Hive metastores

To invoke a Lambda function in your account, you must create a role that has the following permissions:

- AWSLambdaVPCAccessExecutionRole – An AWS Lambda execution role permission to manage elastic network interfaces that connect your function to a VPC. Ensure that you have a sufficient number of network interfaces and IP addresses available.
- AmazonAthenaFullAccess – The AmazonAthenaFullAccess (p. 531) managed policy grants full access to Athena.
- An Amazon S3 policy to allow the Lambda function to write to S3 and to allow Athena to read from S3.

For example, the following policy defines the permission for the spill location s3://mybucket/spill.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetBucketLocation",
        "s3:GetObject",
        "s3:ListBucket",
        "s3:PutObject"
      ],
      "Resource": ["arn:aws:s3:::mybucket/spill"
    ]
  ]
}
```

Whenever you use IAM policies, make sure that you follow IAM best practices. For more information, see Security best practices in IAM in the IAM User Guide.

Creating Lambda functions

To create a Lambda function in your account, function development permissions or the AWSLambdaFullAccess role are required. For more information, see Identity-based IAM policies for AWS Lambda.

Because Athena uses the AWS Serverless Application Repository to create Lambda functions, the superuser or administrator who creates Lambda functions should also have IAM policies to allow Athena federated queries (p. 562).

Catalog registration and metadata API operations

For access to catalog registration API and metadata API operations, use the AmazonAthenaFullAccess managed policy (p. 531). If you do not use this policy, add the following API operations to your Athena policies:
Allow Lambda function access to external Hive metastores

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "athena:ListDataCatalogs",
                "athena:GetDataCatalog",
                "athena:CreateDataCatalog",
                "athena:UpdateDataCatalog",
                "athena:DeleteDataCatalog",
                "athena:GetDatabase",
                "athena:ListDatabases",
                "athena:GetTableMetadata",
                "athena:ListTableMetadata"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

**Cross Region Lambda invocation**

To invoke a Lambda function in a region other than the region in which you are running Athena queries, use the full ARN of the Lambda function. By default, Athena invokes Lambda functions defined in the same region. If you need to invoke a Lambda function to access a Hive metastore in a region other than the region in which you run Athena queries, you must provide the full ARN of the Lambda function.

For example, suppose you define the catalog `ehms` on the Europe (Frankfurt) Region `eu-central-1` to use the following Lambda function in the US East (N. Virginia) Region.

```
```

When you specify the full ARN in this way, Athena can call the `external-hms-service-new` Lambda function on `us-east-1` to fetch the Hive metastore data from `eu-central-1`.

**Note**
The catalog `ehms` should be registered in the same region that you run Athena queries.

**Cross account Lambda invocation**

Sometimes you might require access to a Hive metastore from a different account. For example, to run a Hive metastore, you might launch an EMR cluster from an account that is different from the one that you use for Athena queries. Different groups or teams might run Hive metastore with different accounts inside their VPC. Or you might want to access metadata from different Hive metastores from different groups or teams.

Athena uses the [AWS Lambda support for cross account access](https://docs.aws.amazon.com/lambda/latest/dg/access-control-account-bound.html) to enable cross account access for Hive Metastores.

**Note**
Note that cross account access for Athena normally implies cross account access for both metadata and data in Amazon S3.

Imagine the following scenario:

- Account 111122223333 sets up the Lambda function `external-hms-service-new` on `us-east-1` in Athena to access a Hive Metastore running on an EMR cluster.
- Account 111122223333 wants to allow account 444455556666 to access the Hive Metastore data.
To grant account 444455556666 access to the Lambda function external-hms-service-new, account 111122223333 uses the following AWS CLI add-permission command. The command has been formatted for readability.

```
$ aws --profile perf-test lambda add-permission
   --function-name external-hms-service-new
   --region us-east-1
   --statement-id Id-ehms-invocation2
   --action "lambda:InvokeFunction"
   --principal arn:aws:iam::444455556666:user/perf1-test

   
   
   "Statement": "{
       "Sid": "Id-ehms-invocation2",
       "Effect": "Allow",
       "Principal": {"AWS": "arn:aws:iam::444455556666:user/perf1-test"},
       "Action": "lambda:InvokeFunction",
   }
```

To check the Lambda permission, use the get-policy command, as in the following example. The command has been formatted for readability.

```
$ aws --profile perf-test lambda get-policy
   --region us-east-1

   
   
   "RevisionId": "71le93ea-985f-44c9-a09f-5f2a2829d40f",
   "Policy": "{"Version":"2012-10-17",
   "Id": "default",
   "Statement": [{"Sid": "Id-ehms-invocation2",
   "Effect": "Allow",
   "Principal": {"AWS": "arn:aws:iam::444455556666:user/perf1-test"},
   "Action": "lambda:InvokeFunction",
```

After adding the permission, you can use a full ARN of the Lambda function on us-east-1 like the following when you define catalog ehms:

```
```

For information about cross region invocation, see Cross Region Lambda invocation (p. 560) earlier in this topic.

**Granting cross-account access to data**

Before you can run Athena queries, you must grant cross account access to the data in Amazon S3. You can do this in one of the following ways:

- Update the access control list policy of the Amazon S3 bucket with a canonical user ID.
- Add cross account access to the Amazon S3 bucket policy.

For example, add the following policy to the Amazon S3 bucket policy in the account 111122223333 to allow account 444455556666 to read data from the Amazon S3 location specified.

```
{
   
   
   
   
}
Example IAM permissions policies to allow Athena Federated Query

The permission policy examples in this topic demonstrate required allowed actions and the resources for which they are allowed. Examine these policies carefully and modify them according to your requirements before attaching them to IAM identities.

For information about attaching policies to IAM identities, see Adding and removing IAM identity permissions in the IAM User Guide.

- Example Policy to Allow an IAM Principal to Run and Return Results Using Athena Federated Query (p. 562)
- Example Policy to Allow an IAM Principal to Create a Data Source Connector (p. 564)

Example – Allow an IAM principal to run and return results using Athena Federated Query

The following identity-based permissions policy allows actions that a user or other IAM principal requires to use Athena Federated Query. Principals who are allowed to perform these actions are able to run queries that specify Athena catalogs associated with a federated data source.
Amazon Athena User Guide
Allow access to Athena Federated Query

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "athena:GetWorkGroup",
        "s3:PutObject",
        "s3:GetObject",
        "athena:StartQueryExecution",
        "s3:AbortMultipartUpload",
        "lambda:InvokeFunction",
        "athena:StopQueryExecution",
        "athena:GetQueryExecution",
        "athena:GetQueryResults",
        "s3:ListMultipartUploadParts"
      ],
      "Resource": [
        "arn:aws:athena::*:111122223333:workgroup/WorkgroupName",
        "arn:aws:s3:::MyQueryResultsBucket/*",
        "arn:aws:s3:::MyLambdaSpillBucket/*",
        "arn:aws:lambda:*:111122223333:function:OneAthenaLambdaFunction",
        "arn:aws:lambda:*:111122223333:function:AnotherAthenaLambdaFunction"
      ]
    },
    {
      "Sid": "VisualEditor1",
      "Effect": "Allow",
      "Action": "athena:ListWorkGroups",
      "Resource": "*"
    },
    {
      "Sid": "VisualEditor2",
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket",
        "s3:GetBucketLocation"
      ],
      "Resource": "arn:aws:s3:::MyLambdaSpillBucket"
    }
  ]
}
```

### Explanation of permissions

<table>
<thead>
<tr>
<th>Allowed actions</th>
<th>Explanation</th>
</tr>
</thead>
</table>

`s3:PutObject` and `s3:AbortMultipartUpload` allow writing query results to all sub-folders of the query results bucket as specified by the `arn:aws:s3:::MyQueryResultsBucket/*` resource identifier, where `MyQueryResultsBucket` is the Athena query results bucket. For more information, see Working...
<table>
<thead>
<tr>
<th>Allowed actions</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>s3:GetObject</td>
<td>Allows reading of query results and query history for the resource specified as arn:aws:s3:::MyQueryResultsBucket, where MyQueryResultsBucket is the Athena query results bucket.</td>
</tr>
<tr>
<td>s3:GetObject</td>
<td>Also allows reading from the resource specified as &quot;arn:aws:s3:::MyLambdaSpillBucket/MyLambdaSpillPrefix*&quot;, where MyLambdaSpillPrefix is specified in the configuration of the Lambda function or functions being invoked.</td>
</tr>
<tr>
<td>&quot;lambda:InvokeFunction&quot;</td>
<td>Allows queries to invoke the AWS Lambda functions for the AWS Lambda functions specified in the Resource block. For example, arn:aws:lambda::*:MyAWSAcctId:function:MyAthenaLambdaFunction, where MyAthenaLambdaFunction specifies the name of a Lambda function to be invoked. Multiple functions can be specified as shown in the example.</td>
</tr>
</tbody>
</table>

Example – Allow an IAM principal to create a data source connector

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "VisualEditor0",
         "Effect": "Allow",
         "Action": [
            "lambda:CreateFunction",
            "lambda:ListVersionsByFunction",
            "iam:CreateRole",
            "lambda:GetFunctionConfiguration",
            "iam:AttachRolePolicy",
            "iam:PutRolePolicy",
            "lambda:GetFunctionConcurrency",
            "iam:PassRole",
            "iam:DetachRolePolicy",
            "lambda:ListTags",
            "iam:ListAttachedRolePolicies",
            "iam:DeleteRolePolicy",
            "lambda:DeleteFunction",
            "lambda:GetAlias",
            "iam:ListRolePolicies",
            "iam:GetRole",
            "iam:GetPolicy",
            "lambda:InvokeFunction",
            "lambda:GetFunction",
            "lambda:ListAliases",
            "lambda:UpdateFunctionConfiguration",
            "iam:DeleteRole",
            "lambda:GetFunctionCode",
            "lambda:GetFunctionPermutation",  // This action is not listed in the official AWS IAM documentation, but it is shown in the example.
            "lambda:ListFunctionPermutations", // This action is not listed in the official AWS IAM documentation, but it is shown in the example.
            "lambda:GetFunctionPermutationPolicy", // This action is not listed in the official AWS IAM documentation, but it is shown in the example.
            "lambda:PutFunctionPermutationPolicy", // This action is not listed in the official AWS IAM documentation, but it is shown in the example.
            "lambda:DeleteFunctionPermutation", // This action is not listed in the official AWS IAM documentation, but it is shown in the example.
            "lambda:GetFunctionConcurrency", // This action is not listed in the official AWS IAM documentation, but it is shown in the example.
            "lambda:GetFunctionConcurrency"  // This action is not listed in the official AWS IAM documentation, but it is shown in the example.
         ]
      }
   ]
}
```
Allow access to Athena Federated Query

```json
{
    "Sid": "VisualEditor1",
    "Effect": "Allow",
    "Action": [
        "s3:GetObject",
        "lambda:AddPermission",
        "iam:UpdateRole",
        "lambda:DeleteFunctionConcurrency",
        "lambda:RemovePermission",
        "iam:GetRolePolicy",
        "lambda:GetPolicy"
    ],
    "Resource": [
        "arn:aws:lambda::111122223333:function:MyAthenaLambdaFunctionsPrefix*",
        "arn:aws:s3:::awsserverlessrepo-changesets-1iiv5xa62ln3m/*",
        "arn:aws:iam::*:role/RoleName",
        "arn:aws:iam::111122223333:policy/*"
    ]
},
{
    "Sid": "VisualEditor1",
    "Effect": "Allow",
    "Action": [
        "cloudformation:CreateUploadBucket",
        "cloudformation:DescribeStackDriftDetectionStatus",
        "cloudformation:ListExports",
        "cloudformation:ListStacks",
        "cloudformation:ListImports",
        "lambda:ListFunctions",
        "iam:ListRoles",
        "lambda:GetAccountSettings",
        "ec2:DescribeSecurityGroups",
        "cloudformation:EstimateTemplateCost",
        "ec2:DescribeVpcs",
        "lambda:ListEventSourceMappings",
        "cloudformation:DescribeAccountLimits",
        "ec2:DescribeSubnets",
        "cloudformation:CreateStackSet",
        "cloudformation:ValidateTemplate"
    ],
    "Resource": "*"
},
{
    "Sid": "VisualEditor2",
    "Effect": "Allow",
    "Action": "cloudformation:*",
    "Resource": [
        "arn:aws:cloudformation::111122223333:stack/aws-serverless-repository-MyCFStackPrefix*/*",
        "arn:aws:cloudformation::111122223333:stack/serverlessrepo-MyCFStackPrefix*/*",
        "arn:aws:cloudformation::*:transform/Serverless-*",
        "arn:aws:cloudformation::111122223333:stackset/aws-serverless-repository-MyCFStackPrefix*:*",
        "arn:aws:cloudformation::111122223333:stackset/serverlessrepo-MyCFStackPrefix:*"
    ]
},
{
    "Sid": "VisualEditor3",
    "Effect": "Allow",
    "Action": "serverlessrepo::*",
    "Resource": "arn:aws:serverlessrepo::*:applications/*"
}
}```
## Explanation of permissions

<table>
<thead>
<tr>
<th>Allowed actions</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;lambda:CreateFunction&quot;, &quot;lambda:ListVersionsByFunction&quot;, &quot;lambda:GetFunctionConfiguration&quot;, &quot;lambda:PutFunctionConcurrency&quot;, &quot;lambda:ListTags&quot;, &quot;lambda:DeleteFunction&quot;, &quot;lambda:GetAlias&quot;, &quot;lambda:InvokeFunction&quot;, &quot;lambda:GetFunction&quot;, &quot;lambda:ListAliases&quot;, &quot;lambda:UpdateFunctionConfiguration&quot;, &quot;lambda:UpdateFunctionCode&quot;, &quot;lambda:AddPermission&quot;, &quot;lambda:DeleteFunctionConcurrency&quot;, &quot;lambda:RemovePermission&quot;, &quot;lambda:GetPolicy&quot;</td>
<td>Allow the creation and management of Lambda functions listed as resources. In the example, a name prefix is used in the resource identifier arn:aws:lambda:<em>:MyAWSAcctId:function:MyAthenaLambdaFunctionsPrefix</em>, where MyAthenaLambdaFunctionsPrefix is a shared prefix used in the name of a group of Lambda functions so that they don't need to be specified individually as resources. You can specify one or more Lambda function resources.</td>
</tr>
</tbody>
</table>
| "s3:GetObject" | Allows reading of a bucket that AWS Serverless Application Repository requires as specified by the resource identifier arn:aws:s3:::awsserverlessrepo-changesets-1iiv5xa62ln3m/*.
This bucket may be specific to your account. |
| "cloudformation:*" | Allows the creation and management of AWS CloudFormation stacks specified by the resource MyCFStackPrefix. These stacks and stacksets are how AWS Serverless Application Repository deploys connectors and UDFs. |
| "serverlessrepo:*" | Allows searching, viewing, publishing, and updating applications in the AWS Serverless Application Repository, specified by the resource identifier arn:aws:serverlessrepo:*:applications/*.

## Example IAM permissions policies to allow Amazon Athena User Defined Functions (UDF)

The permission policy examples in this topic demonstrate required allowed actions and the resources for which they are allowed. Examine these policies carefully and modify them according to your requirements before you attach similar permissions policies to IAM identities.

- Example Policy to Allow an IAM Principal to Run and Return Queries that Contain an Athena UDF Statement (p. 567)
- Example Policy to Allow an IAM Principal to Create an Athena UDF (p. 568)
Example – Allow an IAM principal to run and return queries that contain an Athena UDF statement

The following identity-based permissions policy allows actions that a user or other IAM principal requires to run queries that use Athena UDF statements.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "athena:StartQueryExecution",
        "lambda:InvokeFunction",
        "athena:GetQueryResults",
        "s3:ListMultipartUploadParts",
        "athena:GetWorkGroup",
        "s3:PutObject",
        "s3:GetObject",
        "s3:AbortMultipartUpload",
        "athena:StopQueryExecution",
        "athena:GetQueryExecution",
        "s3:GetBucketLocation"
      ],
      "Resource": [
        "arn:aws:athena:*:MyAWSAcctId:workgroup/MyAthenaWorkGroup",
        "arn:aws:s3:::MyQueryResultsBucket/*",
        "arn:aws:lambda:*:MyAWSAcctId:function:OneAthenaLambdaFunction",
        "arn:aws:lambda:*:MyAWSAcctId:function:AnotherAthenaLambdaFunction"
      ]
    },
    {
      "Sid": "VisualEditor1",
      "Effect": "Allow",
      "Action": "athena:ListWorkGroups",
      "Resource": "*"
    }
  ]
}
```

**Explanation of permissions**

<table>
<thead>
<tr>
<th>Allowed actions</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;s3:PutObject&quot;, &quot;s3:GetObject&quot;, &quot;s3:AbortMultipartUpload&quot;</td>
<td>s3:PutObject and s3:AbortMultipartUpload allow writing query results to all sub-folders of the query results bucket as specified by the arn:aws:s3::&lt;MyQueryResultsBucket&gt;/ resource identifier, where <strong>MyQueryResultsBucket</strong> is the Athena query results bucket. For more information, see Working with query results, recent queries, and output files (p. 344).</td>
</tr>
<tr>
<td>Allowed actions</td>
<td>Explanation</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>s3:GetObject</td>
<td>allows reading of query results and query history for the resource specified as arn:aws:s3:::MyQueryResultsBucket, where MyQueryResultsBucket is the Athena query results bucket. For more information, see Working with query results, recent queries, and output files (p. 344). s3:GetObject also allows reading from the resource specified as &quot;arn:aws:s3:::MyLambdaSpillBucket/MyLambdaSpillPrefix&quot;, where MyLambdaSpillPrefix is specified in the configuration of the Lambda function or functions being invoked.</td>
</tr>
<tr>
<td>&quot;lambda:InvokeFunction&quot;</td>
<td>Allows queries to invoke the AWS Lambda functions specified in the Resource block. For example, arn:aws:lambda::*:MyAWSAcctId:function:MyAthenaLambdaFunction, where MyAthenaLambdaFunction specifies the name of a Lambda function to be invoked. Multiple functions can be specified as shown in the example.</td>
</tr>
</tbody>
</table>

Example – Allow an IAM principal to create an Athena UDF

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "lambda:CreateFunction",
        "lambda:ListVersionsByFunction",
        "iam:CreateRole",
        "lambda:GetFunctionConfiguration",
        "iam:AttachRolePolicy",
        "iam:PutRolePolicy",
        "lambda:PutFunctionConcurrency",
        "iam:PassRole",
        "iam:DetachRolePolicy",
        "lambda:ListTags",
        "iam:ListAttachedRolePolicies",
        "iam:DeleteRolePolicy",
        "lambda:DeleteFunction",
        "lambda:GetAlias",
        "iam:ListRolePolicies",
        "iam:GetRole",
        "iam:GetPolicy",
        "lambda:InvokeFunction",
        "lambda:GetFunction",
        "lambda:ListAliases",
        "lambda:UpdateFunctionConfiguration",
        "iam:DeleteRole",
        "lambda:UpdateFunctionCode",
        "s3:GetObject"
      ]
    }
  ]
}
```
"lambda:AddPermission",
"iam:UpdateRole",
"lambda:DeleteFunctionConcurrency",
"lambda:RemovePermission",
"iam:GetRolePolicy",
"lambda:GetPolicy"
],
"Resource": [
"arn:aws:lambda::*:function:MyAthenaLambdaFunctionsPrefix*",
"arn:aws:s3:::awsserverlessrepo-changesets-111122223333/*",
"arn:aws:iam::*:role/RoleName",
"arn:aws:iam::111122223333:policy/*"
]
},
{
"Sid": "VisualEditor1",
"Effect": "Allow",
"Action": [
"cloudformation:CreateUploadBucket",
"cloudformation:DescribeStackDriftDetectionStatus",
"cloudformation:ListExports",
"cloudformation:ListStacks",
"cloudformation:ListImports",
"lambda:ListFunctions",
"iam:ListRoles",
"lambda:GetAccountSettings",
"ec2:DescribeSecurityGroups",
"cloudformation:EstimateTemplateCost",
"ec2:DescribeVpcs",
"lambda:ListEventSourceMappings",
"cloudformation:DescribeAccountLimits",
"ec2:DescribeSubnets",
"cloudformation:CreateStackSet",
"cloudformation:ValidateTemplate"
],
"Resource": "*

"Sid": "VisualEditor2",
"Effect": "Allow",
"Action": "cloudformation:*

"Resource": ["arn:aws:cloudformation::*:stack/aws-serverless-repository-MyCFStackPrefix**",
"arn:aws:cloudformation::*:stack/serverlessrepo-MyCFStackPrefix**",
"arn:aws:cloudformation::*:transform/Serverless-*",
"arn:aws:cloudformation::*:stackset/aws-serverless-repository-MyCFStackPrefix**",
"arn:aws:cloudformation::*:stackset/serverlessrepo-MyCFStackPrefix**"
]
},
{
"Sid": "VisualEditor3",
"Effect": "Allow",
"Action": "serverlessrepo:*

"Resource": "arn:aws:serverlessrepo:*:applications/**
]
## Explanation of permissions

<table>
<thead>
<tr>
<th>Allowed actions</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;lambda:CreateFunction&quot;, &quot;lambda:ListVersionsByFunction&quot;,</td>
<td>Allow the creation and management of Lambda functions listed as resources. In the example, a name prefix is used in the resource identifier arn:aws:lambda:*:MyAWSAcctId:function:MyAthenaLambdaFunctionsPrefix where <strong>MyAthenaLambdaFunctionsPrefix</strong> is a shared prefix used in the name of a group of Lambda functions so that they don't need to be specified individually as resources. You can specify one or more Lambda function resources.</td>
</tr>
<tr>
<td>&quot;lambda:GetFunctionConfiguration&quot;, &quot;lambda:PutFunctionConcurrency&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;lambda:ListTags&quot;, &quot;lambda:DeleteFunction&quot;, &quot;lambda:GetAlias&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;lambda:InvokeFunction&quot;, &quot;lambda:GetFunction&quot;, &quot;lambda:ListAliases&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;lambda:UpdateFunctionConfiguration&quot;, &quot;lambda:UpdateFunctionCode&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;lambda:AddPermission&quot;, &quot;lambda:DeleteFunctionConcurrency&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;lambda:RemovePermission&quot;, &quot;lambda:GetPolicy&quot;, &quot;lambda:GetAccountSettings&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;lambda:GetFunction&quot;, &quot;lambda:ListFunctions&quot;, &quot;lambda:ListEventSourceMappings&quot;,</td>
<td></td>
</tr>
</tbody>
</table>
| "s3:GetObject"                                                                 | Allows reading of a bucket that AWS Serverless Application Repository requires as specified by the resource identifier arn:aws:s3:::awsserverlessrepo-changesets-1iiv3xa62ln3m/*.

| "cloudformation:*"                                                           | Allows the creation and management of AWS CloudFormation stacks specified by the resource **MyCFStackPrefix**. These stacks and stacksets are how AWS Serverless Application Repository deploys connectors and UDFs.                                                                                                                                                                                                                           |
|                                                                              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| "serverlessrepo:*"                                                           | Allows searching, viewing, publishing, and updating applications in the AWS Serverless Application Repository, specified by the resource identifier arn:aws:serverlessrepo::*:applications/*.

## Allowing access for ML with Athena

IAM principals who run Athena ML queries must be allowed to perform the sagemaker:invokeEndpoint action for Sagemaker endpoints that they use. Include a policy statement similar to the following in identity-based permissions policies attached to user identities. In addition, attach the AWS managed policy: AmazonAthenaFullAccess (p. 531), which grants full access to Athena actions, or a modified inline policy that allows a subset of actions.

Replace arn:aws:sagemaker:region:AWSAcctID:ModelEndpoint in the example with the ARN or ARNs of model endpoints to be used in queries. For more information, see Actions, resources, and condition keys for SageMaker in the Service Authorization Reference.

```json
{
  "Effect": "Allow",
  "Action": [
```
Enabling federated access to the Athena API

This section discusses federated access that allows a user or client application in your organization to call Amazon Athena API operations. In this case, your organization's users don't have direct access to Athena. Instead, you manage user credentials outside of AWS in Microsoft Active Directory. Active Directory supports SAML 2.0 (Security Assertion Markup Language 2.0).

To authenticate users in this scenario, use the JDBC or ODBC driver with SAML 2.0 support to access Active Directory Federation Services (ADFS) 3.0 and enable a client application to call Athena API operations.

For more information about SAML 2.0 support on AWS, see About SAML 2.0 federation in the IAM User Guide.

Note

Federated access to the Athena API is supported for a particular type of identity provider (IdP), the Active Directory Federation Service (ADFS 3.0), which is part of Windows Server. Access is established through the versions of JDBC or ODBC drivers that support SAML 2.0. For information, see Using Athena with the JDBC driver (p. 183) and Connecting to Amazon Athena with ODBC (p. 185).

Topics

- Before you begin (p. 571)
- Architecture diagram (p. 571)
- Procedure: SAML-based federated access to the Athena API (p. 572)

Before you begin

Before you begin, complete the following prerequisites:

- Inside your organization, install and configure the ADFS 3.0 as your IdP.
- Install and configure the latest available versions of JDBC or ODBC drivers on clients that are used to access Athena. The driver must include support for federated access compatible with SAML 2.0. For information, see Using Athena with the JDBC driver (p. 183) and Connecting to Amazon Athena with ODBC (p. 185).

Architecture diagram

The following diagram illustrates this process.
1. A user in your organization uses a client application with the JDBC or ODBC driver to request authentication from your organization's IdP. The IdP is ADFS 3.0.

2. The IdP authenticates the user against Active Directory, which is your organization's Identity Store.

3. The IdP constructs a SAML assertion with information about the user and sends the assertion to the client application via the JDBC or ODBC driver.

4. The JDBC or ODBC driver calls the AWS Security Token Service AssumeRoleWithSAML API operation, passing it the following parameters:
   - The ARN of the SAML provider
   - The ARN of the role to assume
   - The SAML assertion from the IdP

   For more information, see AssumeRoleWithSAML, in the AWS Security Token Service API Reference.

5. The API response to the client application via the JDBC or ODBC driver includes temporary security credentials.

6. The client application uses the temporary security credentials to call Athena API operations, allowing your users to access Athena API operations.

Procedure: SAML-based federated access to the Athena API

This procedure establishes trust between your organization's IdP and your AWS account to enable SAML-based federated access to the Amazon Athena API operation.

To enable federated access to the Athena API:

1. In your organization, register AWS as a service provider (SP) in your IdP. This process is known as relying party trust. For more information, see Configuring your SAML 2.0 IdP with relying party trust in the IAM User Guide. As part of this task, perform these steps:
   a. Obtain the sample SAML metadata document from this URL: https://signin.aws.amazon.com/static/saml-metadata.xml.
b. In your organization's IdP (ADFS), generate an equivalent metadata XML file that describes your
IdP as an identity provider to AWS. Your metadata file must include the issuer name, creation
date, expiration date, and keys that AWS uses to validate authentication responses (assertions)
from your organization.

2. In the IAM console, create a SAML identity provider entity. For more information, see Creating SAML
identity providers in the IAM User Guide. As part of this step, do the following:

b. Upload the SAML metadata document produced by the IdP (ADFS) in Step 1 in this procedure.

3. In the IAM console, create one or more IAM roles for your IdP. For more information, see Creating a
role for a third-party Identity Provider (federation) in the IAM User Guide. As part of this step, do the
following:

- In the role's permission policy, list actions that users from your organization are allowed to do in
  AWS.
- In the role's trust policy, set the SAML provider entity that you created in Step 2 of this procedure
  as the principal.

This establishes a trust relationship between your organization and AWS.

4. In your organization's IdP (ADFS), define assertions that map users or groups in your organization to
the IAM roles. The mapping of users and groups to the IAM roles is also known as a claim rule. Note
that different users and groups in your organization might map to different IAM roles.

For information about configuring the mapping in ADFS, see the blog post: Enabling federation to
AWS using Windows Active Directory, ADFS, and SAML 2.0.

5. Install and configure the JDBC or ODBC driver with SAML 2.0 support. For information, see Using
Athena with the JDBC driver (p. 183) and Connecting to Amazon Athena with ODBC (p. 185).

6. Specify the connection string from your application to the JDBC or ODBC driver. For information
about the connection string that your application should use, see the topic "Using the Active
Directory Federation Services (ADFS) Credentials Provider" in the JDBC Driver Installation and
Configuration Guide, or a similar topic in the ODBC Driver Installation and Configuration Guide
available as PDF downloads from the Using Athena with the JDBC driver (p. 183) and Connecting to
Amazon Athena with ODBC (p. 185) topics.

Following is a high-level summary of configuring the connection string to the drivers:

1. In the AwsCredentialsProviderClass configuration, set the
   com.simba.athena.iamsupport.plugin.AdfsCredentialsProvider to indicate that you
   want to use SAML 2.0 based authentication via ADFS IdP.
2. For idp_host, provide the host name of the ADFS IdP server.
3. For idp_port, provide the port number that the ADFS IdP listens on for the SAML assertion
   request.
4. For UID and PwD, provide the AD domain user credentials. When using the driver on Windows,
   if UID and PwD are not provided, the driver attempts to obtain the user credentials of the user
   logged in to the Windows machine.
5. Optionally, set ssl_insecure to true. In this case, the driver does not check the authenticity
   of the SSL certificate for the ADFS IdP server. Setting to true is needed if the ADFS IdP's SSL
   certificate has not been configured to be trusted by the driver.
6. To enable mapping of an Active Directory domain user or group to one or more IAM roles
   (as mentioned in step 4 of this procedure), in the preferred_role for the JDBC or ODBC
   connection, specify the IAM role (ARN) to assume for the driver connection. Specifying the
   preferred_role is optional, and is useful if the role is not the first role listed in the claim rule.
As a result of this procedure, the following actions occur:

1. The JDBC or ODBC driver calls the AWS STS AssumeRoleWithSAML API, and passes it the assertions, as shown in step 4 of the architecture diagram (p. 571).

2. AWS makes sure that the request to assume the role comes from the IdP referenced in the SAML provider entity.

3. If the request is successful, the AWS STS AssumeRoleWithSAML API operation returns a set of temporary security credentials, which your client application uses to make signed requests to Athena.

Your application now has information about the current user and can access Athena programmatically.

Logging and monitoring in Athena

To detect incidents, receive alerts when incidents occur, and respond to them, use these options with Amazon Athena:

- **Monitor Athena with AWS CloudTrail** – AWS CloudTrail provides a record of actions taken by a user, role, or an AWS service in Athena. It captures calls from the Athena console and code calls to the Athena API operations as events. This allows you to determine the request that was made to Athena, the IP address from which the request was made, who made the request, when it was made, and additional details. For more information, see Logging Amazon Athena API calls with AWS CloudTrail (p. 575).

You can also use Athena to query the CloudTrail log files not only for Athena, but for other AWS services. For more information, see Querying AWS CloudTrail logs (p. 454).

- **Monitor Athena usage with CloudTrail and Amazon QuickSight** – Amazon QuickSight is a fully managed, cloud-powered business intelligence service that lets you create interactive dashboards your organization can access from any device. For an example of a solution that uses CloudTrail and Amazon QuickSight to monitor Athena usage, see the AWS Big Data blog post How Realtor.com monitors Amazon Athena usage with AWS CloudTrail and Amazon QuickSight.

- **Use CloudWatch Events with Athena** – CloudWatch Events delivers a near real-time stream of system events that describe changes in AWS resources. CloudWatch Events becomes aware of operational changes as they occur, responds to them, and takes corrective action as necessary, by sending messages to respond to the environment, activating functions, making changes, and capturing state information. Events are emitted on a best effort basis. To use CloudWatch Events with Athena, create a rule that triggers on an Athena API call via CloudTrail. For more information, see Creating a CloudWatch Events rule that triggers on an AWS API call using CloudTrail in the Amazon CloudWatch Events User Guide.

- **Use workgroups to separate users, teams, applications, or workloads, and to set query limits and control query costs** – You can view query-related metrics in Amazon CloudWatch, control query costs by configuring limits on the amount of data scanned, create thresholds, and trigger actions, such as Amazon SNS alarms, when these thresholds are breached. For a high-level procedure, see Setting up workgroups (p. 631). Use resource-level IAM permissions to control access to a specific workgroup. For more information, see Using workgroups for running queries (p. 629) and Controlling costs and monitoring queries with CloudWatch metrics and events (p. 647).

Topics

- Logging Amazon Athena API calls with AWS CloudTrail (p. 575)
Logging Amazon Athena API calls with AWS CloudTrail

Athena is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Athena.

CloudTrail captures all API calls for Athena as events. The calls captured include calls from the Athena console and code calls to the Athena API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for Athena. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in Event history.

Using the information collected by CloudTrail, you can determine the request that was made to Athena, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, see the AWS CloudTrail User Guide.

You can use Athena to query CloudTrail log files from Athena itself and from other AWS services. For more information, see Querying AWS CloudTrail logs (p. 454), Hive JSON SerDe (p. 320), and the AWS Big Data Blog post Use CTAS statements with Amazon Athena to reduce cost and improve performance, which uses CloudTrail to provide insight into Athena usage.

Athena information in CloudTrail

CloudTrail is enabled on your Amazon Web Services account when you create the account. When activity occurs in Athena, that activity is recorded in a CloudTrail event along with other AWS service events in Event history. You can view, search, and download recent events in your Amazon Web Services account. For more information, see Viewing events with CloudTrail event history.

For an ongoing record of events in your Amazon Web Services account, including events for Athena, create a trail. A trail enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- Overview for creating a trail
- CloudTrail supported services and integrations
- Configuring Amazon SNS notifications for CloudTrail
- Receiving CloudTrail log files from multiple regions and Receiving CloudTrail log files from multiple accounts

All Athena actions are logged by CloudTrail and are documented in the Amazon Athena API Reference. For example, calls to the StartQueryExecution and GetQueryResults actions generate entries in the CloudTrail log files.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or AWS Identity and Access Management (IAM) user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity element.
Understanding Athena log file entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so they don't appear in any specific order.

The following examples demonstrate CloudTrail log entries for:

- StartQueryExecution (Successful) (p. 576)
- StartQueryExecution (Failed) (p. 576)
- CreateNamedQuery (p. 577)

StartQueryExecution (successful)

```json
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "IAMUser",
    "principalId": "EXAMPLE_PRINCIPAL_ID",
    "arn": "arn:aws:iam::123456789012:user/johndoe",
    "accountId": "123456789012",
    "accessKeyId": "EXAMPLE_KEY_ID",
    "userName": "johndoe"
  },
  "eventTime": "2017-05-04T00:23:55Z",
  "eventSource": "athena.amazonaws.com",
  "eventName": "StartQueryExecution",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "77.88.999.69",
  "userAgent": "aws-internal/3",
  "requestParameters": {
    "clientRequestToken": "16bc6e70-f972-4260-b18a-db1b623cb35c",
    "resultConfiguration": {
      "outputLocation": "s3://athena-johndoe-test/test/"
    },
    "queryString": "Select 10"
  },
  "responseElements": {
    "queryExecutionId": "b621c254-74e0-48e3-9630-78ed857782f9"
  },
  "requestID": "f5039b01-305f-11e7-b146-c3fc56a7dc7a",
  "eventID": "c97cf8c8-6112-467a-8777-53bb38f83fd5",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```

StartQueryExecution (failed)

```json
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "IAMUser",
    "principalId": "EXAMPLE_PRINCIPAL_ID",
    "arn": "arn:aws:iam::123456789012:user/johndoe",
    "accountId": "123456789012",
    "accessKeyId": "EXAMPLE_KEY_ID",
    "userName": "johndoe"
  },
  "eventTime": "2017-05-04T00:23:55Z",
  "eventSource": "athena.amazonaws.com",
  "eventName": "StartQueryExecution",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "77.88.999.69",
  "userAgent": "aws-internal/3",
  "requestParameters": {
    "clientRequestToken": "16bc6e70-f972-4260-b18a-db1b623cb35c",
    "resultConfiguration": {
      "outputLocation": "s3://athena-johndoe-test/test/"
    },
    "queryString": "Select 10"
  },
  "responseElements": {
    "queryExecutionId": "b621c254-74e0-48e3-9630-78ed857782f9"
  },
  "requestID": "f5039b01-305f-11e7-b146-c3fc56a7dc7a",
  "eventID": "c97cf8c8-6112-467a-8777-53bb38f83fd5",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```
Compliance validation for Amazon Athena

Third-party auditors assess the security and compliance of Amazon Athena as part of multiple AWS compliance programs. These include SOC, PCI, FedRAMP, and others.

For a list of AWS services in scope of specific compliance programs, see AWS services in scope by compliance program. For general information, see AWS compliance programs.
You can download third-party audit reports using AWS Artifact. For more information, see Downloading reports in AWS Artifact.

Your compliance responsibility when using Athena is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- **Security and compliance quick start guides** – These deployment guides discuss architectural considerations and provide steps for deploying security- and compliance-focused baseline environments on AWS.
- **Architecting for HIPAA security and compliance whitepaper** – This whitepaper describes how companies can use AWS to create HIPAA-compliant applications.
- **AWS compliance resources** – This collection of workbooks and guides might apply to your industry and location.
- **AWS Config** – This AWS service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- **AWS Security Hub** – This AWS service provides a comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.

## Resilience in Athena

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see [AWS global infrastructure](#).

In addition to the AWS global infrastructure, Athena offers several features to help support your data resiliency and backup needs.

Athena is serverless, so there is no infrastructure to set up or manage. Athena is highly available and runs queries using compute resources across multiple Availability Zones, automatically routing queries appropriately if a particular Availability Zone is unreachable. Athena uses Amazon S3 as its underlying data store, making your data highly available and durable. Amazon S3 provides durable infrastructure to store important data and is designed for durability of 99.999999999% of objects. Your data is redundantly stored across multiple facilities and multiple devices in each facility.

## Infrastructure security in Athena

As a managed service, Amazon Athena is protected by the AWS global network security procedures that are described in the [Amazon Web Services: Overview of security processes](#) whitepaper.

You use AWS published API calls to access Athena through the network. Clients must support TLS (Transport Layer Security) 1.0. We recommend TLS 1.2 or later. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). Most modern systems such as Java 7 and later support these modes. Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the [AWS Security Token Service](#) (AWS STS) to generate temporary security credentials to sign requests.
Use IAM policies to restrict access to Athena operations. Whenever you use IAM policies, make sure that you follow IAM best practices. For more information, see Security best practices in IAM in the IAM User Guide.

Athena managed policies (p. 530) are easy to use, and are automatically updated with the required actions as the service evolves. Customer-managed and inline policies allow you to fine-tune policies by specifying more granular Athena actions within the policy. Grant appropriate access to the Amazon S3 location of the data. For detailed information and scenarios about how to grant Amazon S3 access, see Example walkthroughs: Managing access in the Amazon Simple Storage Service Developer Guide. For more information and an example of which Amazon S3 actions to allow, see the example bucket policy in Cross-Account Access (p. 537).

Topics
- Connect to Amazon Athena using an interface VPC endpoint (p. 579)

Connect to Amazon Athena using an interface VPC endpoint

You can improve the security posture of your VPC by using an interface VPC endpoint (AWS PrivateLink) and an AWS Glue VPC endpoint in your Virtual Private Cloud (VPC). An interface VPC endpoint improves security by giving you control over what destinations can be reached from inside your VPC. Each VPC endpoint is represented by one or more Elastic network interfaces (ENIs) with private IP addresses in your VPC subnets.

The interface VPC endpoint connects your VPC directly to Athena without an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. The instances in your VPC don’t need public IP addresses to communicate with the Athena API.

To use Athena through your VPC, you must connect from an instance that is inside the VPC or connect your private network to your VPC by using an Amazon Virtual Private Network (VPN) or AWS Direct Connect. For information about Amazon VPN, see VPN connections in the Amazon Virtual Private Cloud User Guide. For information about AWS Direct Connect, see Creating a connection in the AWS Direct Connect User Guide.

Athena supports VPC endpoints in all AWS Regions where both Amazon VPC and Athena are available.

You can create an interface VPC endpoint to connect to Athena using the AWS Management Console or AWS Command Line Interface (AWS CLI) commands. For more information, see Creating an interface endpoint.

After you create an interface VPC endpoint, if you enable private DNS hostnames for the endpoint, the default Athena endpoint (https://athena.Region.amazonaws.com) resolves to your VPC endpoint.

If you do not enable private DNS hostnames, Amazon VPC provides a DNS endpoint name that you can use in the following format:

```
VPC_Endpoint_ID.athena.Region.vpce.amazonaws.com
```

For more information, see Interface VPC endpoints (AWS PrivateLink) in the Amazon VPC User Guide.

Athena supports making calls to all of its API actions inside your VPC.

Create a VPC endpoint policy for Athena

You can create a policy for Amazon VPC endpoints for Athena to specify the following:
• The principal that can perform actions.
• The actions that can be performed.
• The resources on which actions can be performed.

For more information, see Controlling access to services with VPC endpoints in the Amazon VPC User Guide.

Whenever you use IAM policies, make sure that you follow IAM best practices. For more information, see Security best practices in IAM in the IAM User Guide.

Example – VPC endpoint policy for Athena actions

The endpoint to which this policy is attached grants access to the listed athena actions to all principals in workgroupA.

```json
{
    "Statement": [{
        "Principal": "*",
        "Effect": "Allow",
        "Action": [
            "athena:StartQueryExecution",
            "athena:RunQuery",
            "athena:GetQueryExecution",
            "athena:GetQueryResults",
            "athena:StopQueryExecution",
            "athena:ListWorkGroups",
            "athena:GetWorkGroup",
            "athena:TagResource"
        ],
        "Resource": [
            "arn:aws:athena:us-west-1:AWSAccountId:workgroup/workgroupA"
        ]
    }]
}
```

Configuration and vulnerability analysis in Athena

Athena is serverless, so there is no infrastructure to set up or manage. AWS handles basic security tasks, such as guest operating system (OS) and database patching, firewall configuration, and disaster recovery. These procedures have been reviewed and certified by the appropriate third parties. For more details, see the following AWS resources:

• Shared responsibility model
• Best practices for security, identity, & compliance

Using Athena to query data registered with AWS Lake Formation

AWS Lake Formation allows you to define and enforce database, table, and column-level access policies when using Athena queries to read data stored in Amazon S3. Lake Formation provides an authorization
and governance layer on data stored in Amazon S3. You can use a hierarchy of permissions in Lake Formation to grant or revoke permissions to read data catalog objects such as databases, tables, and columns. Lake Formation simplifies the management of permissions and allows you to implement fine-grained access control (FGAC) for your data.

You can use Athena to query both data that is registered with Lake Formation and data that is not registered with Lake Formation.

Lake Formation permissions apply when using Athena to query source data from Amazon S3 locations that are registered with Lake Formation. Lake Formation permissions also apply when you create databases and tables that point to registered Amazon S3 data locations. To use Athena with data registered using Lake Formation, Athena must be configured to use the AWS Glue Data Catalog.

Lake Formation permissions do not apply when writing objects to Amazon S3, nor do they apply when querying data stored in Amazon S3 or metadata that is not registered with Lake Formation. For source data in Amazon S3 and metadata that is not registered with Lake Formation, access is determined by IAM permissions policies for Amazon S3 and AWS Glue actions. Athena query results locations in Amazon S3 cannot be registered with Lake Formation, and IAM permissions policies for Amazon S3 control access. In addition, Lake Formation permissions do not apply to Athena query history. You can use Athena workgroups to control access to query history.

For more information about Lake Formation, see Lake Formation FAQs and the AWS Lake Formation Developer Guide.

Topics
- How Athena accesses data registered with Lake Formation (p. 581)
- Considerations and limitations when using Athena to query data registered with Lake Formation (p. 583)
- Managing Lake Formation and Athena user permissions (p. 586)
- Applying Lake Formation permissions to existing databases and tables (p. 588)
- Using Lake Formation and the Athena JDBC and ODBC drivers for federated access to Athena (p. 589)

## How Athena accesses data registered with Lake Formation

The access workflow described in this section applies only when running Athena queries on Amazon S3 locations and metadata objects that are registered with Lake Formation. For more information, see Registering a data lake in the AWS Lake Formation Developer Guide. In addition to registering data, the Lake Formation administrator applies Lake Formation permissions that grant or revoke access to metadata in the Data Catalog and the data location in Amazon S3. For more information, see Security and access control to metadata and data in the AWS Lake Formation Developer Guide.

Each time an Athena principal (user, group, or role) runs a query on data registered using Lake Formation, Lake Formation verifies that the principal has the appropriate Lake Formation permissions to the database, table, and Amazon S3 location as appropriate for the query. If the principal has access, Lake Formation *vends* temporary credentials to Athena, and the query runs.

The following diagram illustrates the flow described above.
The following diagram shows how credential vending works in Athena on a query-by-query basis for a hypothetical SELECT query on a table with an Amazon S3 location registered in Lake Formation:

1. A principal runs a SELECT query in Athena.
2. Athena analyzes the query and checks Lake Formation permissions to see if the principal has been granted access to the table and table columns.
3. If the principal has access, Athena requests credentials from Lake Formation. If the principal does not have access, Athena issues an access denied error.
4. Lake Formation issues credentials to Athena to use when reading data from Amazon S3, along with the list of allowed columns.
5. Athena uses the Lake Formation temporary credentials to query the data from Amazon S3. After the query completes, Athena discards the credentials.
Considerations and limitations when using Athena to query data registered with Lake Formation

Consider the following when using Athena to query data registered in Lake Formation. For additional information, see Known issues for AWS Lake Formation in the AWS Lake Formation Developer Guide.

Considerations and Limitations

- Column metadata visible to unauthorized users in some circumstances with Avro and custom SerDe (p. 583)
- Working with Lake Formation permissions to views (p. 583)
- Lake Formation fine-grained access control and Athena workgroups (p. 584)
- Lake Formation data filter limitations (p. 584)
- Athena query results location in Amazon S3 not registered with Lake Formation (p. 584)
- Use Athena workgroups to limit access to query history (p. 584)
- Cross-account Data Catalog access (p. 584)
- CSE-KMS encrypted Amazon S3 locations registered with Lake Formation cannot be queried in Athena (p. 586)
- Partitioned data locations registered with Lake Formation must be in table subdirectories (p. 586)
- Create table as select (CTAS) queries require Amazon S3 write permissions (p. 586)
- The DESCRIBE permission is required on the default database (p. 586)

Column metadata visible to unauthorized users in some circumstances with Avro and custom SerDe

Lake Formation column-level authorization prevents users from accessing data in columns for which the user does not have Lake Formation permissions. However, in certain situations, users are able to access metadata describing all columns in the table, including the columns for which they do not have permissions to the data.

This occurs when column metadata is stored in table properties for tables using either the Apache Avro storage format or using a custom Serializer/Deserializer (SerDe) in which table schema is defined in table properties along with the SerDe definition. When using Athena with Lake Formation, we recommend that you review the contents of table properties that you register with Lake Formation and, where possible, limit the information stored in table properties to prevent any sensitive metadata from being visible to users.

Working with Lake Formation permissions to views

For data registered with Lake Formation, an Athena user can create a VIEW only if they have Lake Formation permissions to the tables, columns, and source Amazon S3 data locations on which the VIEW is based. After a VIEW is created in Athena, Lake Formation permissions can be applied to the VIEW. Column-level permissions are not available for a VIEW. Users who have Lake Formation permissions to a VIEW but do not have permissions to the table and columns on which the view was based are not able to use the VIEW to query data. However, users with this mix of permissions are able to use statements like DESCRIBE VIEW, SHOW CREATE VIEW, and SHOW COLUMNS to see VIEW metadata. For this reason, be sure to align Lake Formation permissions for each VIEW with underlying table permissions. Cell filters defined on a table do not apply to a VIEW for that table. For more information about setting up permissions for shared views across accounts, see Cross-account Data Catalog access (p. 584).
Lake Formation fine-grained access control and Athena workgroups

Users in the same Athena workgroup can see the data that Lake Formation fine-grained access control has configured to be accessible to the workgroup. For more information about using fine-grained access control in Lake Formation, see Manage fine-grained access control using AWS Lake Formation in the AWS Big Data Blog.

Lake Formation data filter limitations

Lake Formation data filters cannot be used with partition projection in Athena. If you enable data filters on a database or table in Lake Formation, partition projection stops working.

Lake Formation data filters do not support the Hive JSON SerDe.

Athena query results location in Amazon S3 not registered with Lake Formation

The query results locations in Amazon S3 for Athena cannot be registered with Lake Formation. Lake Formation permissions do not limit access to these locations. Unless you limit access, Athena users can access query result files and metadata when they do not have Lake Formation permissions for the data. To avoid this, we recommend that you use workgroups to specify the location for query results and align workgroup membership with Lake Formation permissions. You can then use IAM permissions policies to limit access to query results locations. For more information about query results, see Working with query results, recent queries, and output files (p. 344).

Use Athena workgroups to limit access to query history

Athena query history exposes a list of saved queries and complete query strings. Unless you use workgroups to separate access to query histories, Athena users who are not authorized to query data in Lake Formation are able to view query strings run on that data, including column names, selection criteria, and so on. We recommend that you use workgroups to separate query histories, and align Athena workgroup membership with Lake Formation permissions to limit access. For more information, see Using workgroups to control query access and costs (p. 629).

Cross-account Data Catalog access

To access a data catalog in another account, you can use Athena's cross-account AWS Glue feature or set up cross-account access in Lake Formation.

Athena cross-account Data Catalog access

You can use Athena's cross-account AWS Glue catalog feature to register the catalog in your account. This capability is available only in Athena engine version 2 and later versions and is limited to same-Region use between accounts. For more information, see Registering an AWS Glue Data Catalog from another account (p. 24).

If the Data Catalog to be shared has a resource policy configured in AWS Glue, it must be updated to allow access to the AWS Resource Access Manager and grant permissions to Account B to use Account A's Data Catalog, as in the following example.

```json
{
   "Version": "2012-10-17",
```
Considerations and limitations

“Statement”:

```
{"Statement": [{
  "Effect": "Allow",
  "Principal": {
    "Service": "ram.amazonaws.com"
  },
  "Action": "glue:ShareResource",
  "Resource": [
  ]},
  "Effect": "Allow",
  "Principal": {
    "AWS": "arn:aws:iam::<ACCOUNT-B>:root"
  },
  "Action": "glue:*",
  "Resource": [
  ]}
]}
```

For more information, see Cross-account access to AWS Glue data catalogs (p. 548).

Setting up cross-account access in Lake Formation

AWS Lake Formation lets you use a single account to manage a central Data Catalog. You can use this feature to implement cross-account access to Data Catalog metadata and underlying data. For example, an owner account can grant another (recipient) account SELECT permission on a table.

For a shared database or table to appear in the Athena Query Editor, you create a resource link in Lake Formation to the shared database or table. When the recipient account in Lake Formation queries the owner's table, CloudTrail adds the data access event to the logs for both the recipient account and the owner account.

For shared views, keep in mind the following points:

- Queries are run on target resource links, not on the source table or view, and then the output is shared to the target account.
- It is not sufficient to share only the view. All the tables that are involved in creating the view must be part of the cross-account share.
- The name of the resource link created on the shared resources must match the name of the resource in the owner account. If the name does not match, an error message like Failed analyzing stored view 'awsdatacatalog.my-lf-resource-link.my-lf-view': line 3:3: Schema schema_name does not exist occurs.

For more information about cross-account access in Lake Formation, see the following resources in the AWS Lake Formation Developer Guide:

- Cross-account access
- How resource links work in Lake Formation
- Cross-account CloudTrail logging
CSE-KMS encrypted Amazon S3 locations registered with Lake Formation cannot be queried in Athena

Amazon S3 data locations that are registered with Lake Formation and encrypted using client-side encryption (CSE) with AWS KMS customer-managed keys (CSE-KMS) cannot be queried using Athena. You still can use Athena to query CSE-KMS encrypted Amazon S3 data locations that are not registered with Lake Formation and use IAM policies to allow or deny access.

Partitioned data locations registered with Lake Formation must be in table subdirectories

Partitioned tables registered with Lake Formation must have partitioned data in directories that are subdirectories of the table in Amazon S3. For example, a table with the location s3://mydata/mytable and partitions s3://mydata/mytable/dt=2019-07-11, s3://mydata/mytable/dt=2019-07-12, and so on can be registered with Lake Formation and queried using Athena. On the other hand, a table with the location s3://mydata/mytable and partitions located in s3://mydata/dt=2019-07-11, s3://mydata/dt=2019-07-12, and so on, cannot be registered with Lake Formation. Because such partitions are not subdirectories of s3://mydata/mytable, they also cannot be read from Athena.

Create table as select (CTAS) queries require Amazon S3 write permissions

Create Table As Statements (CTAS) require write access to the Amazon S3 location of tables. To run CTAS queries on data registered with Lake Formation, Athena users must have IAM permissions to write to the table Amazon S3 locations in addition to the appropriate Lake Formation permissions to read the data locations. For more information, see Creating a table from query results (CTAS) (p. 281).

The DESCRIBE permission is required on the default database

The Lake Formation DESCRIBE permission is required on the default database. The following example AWS CLI command grants the DESCRIBE permission on the default database to the user datalake_user1 in AWS account 111122223333.

```
aws lakeformation grant-permissions --principal DataLakePrincipalIdentifier=arn:aws:iam::111122223333:user/datalake_user1 --permissions "DESCRIBE" --resource '{ "Database": {"Name":"default"}}
```

For more information, see the Lake Formation permissions reference in the AWS Lake Formation Developer Guide.

Managing Lake Formation and Athena user permissions

Lake Formation vends credentials to query Amazon S3 data stores that are registered with Lake Formation. If you previously used IAM policies to allow or deny permissions to read data locations in Amazon S3, you can use Lake Formation permissions instead. However, other IAM permissions are still required.

Whenever you use IAM policies, make sure that you follow IAM best practices. For more information, see Security best practices in IAM in the IAM User Guide.
The following sections summarize the permissions required to use Athena to query data registered in Lake Formation. For more information, see Security in AWS Lake Formation in the AWS Lake Formation Developer Guide.

Permissions Summary
- Identity-based permissions for Lake Formation and Athena (p. 587)
- Amazon S3 permissions for Athena query results locations (p. 587)
- Athena workgroup memberships to query history (p. 587)
- Lake Formation permissions to data (p. 588)
- IAM permissions to write to Amazon S3 locations (p. 588)
- Permissions to encrypted data, metadata, and Athena query results (p. 588)
- Resource-based permissions for Amazon S3 buckets in external accounts (optional) (p. 588)

Identity-based permissions for Lake Formation and Athena

Anyone using Athena to query data registered with Lake Formation must have an IAM permissions policy that allows the `lakeformation:GetDataAccess` action. The AWS managed policy: `AmazonAthenaFullAccess` allows this action. If you use inline policies, be sure to update permissions policies to allow this action.

In Lake Formation, a data lake administrator has permissions to create metadata objects such as databases and tables, grant Lake Formation permissions to other users, and register new Amazon S3 locations. To register new locations, permissions to the service-linked role for Lake Formation are required. For more information, see Create a data lake administrator and Service-linked role permissions for Lake Formation in the AWS Lake Formation Developer Guide.

An Lake Formation user can use Athena to query databases, tables, table columns, and underlying Amazon S3 data stores based on Lake Formation permissions granted to them by data lake administrators. Users cannot create databases or tables, or register new Amazon S3 locations with Lake Formation. For more information, see Create a data lake user in the AWS Lake Formation Developer Guide.

In Athena, identity-based permissions policies, including those for Athena workgroups, still control access to Athena actions for Amazon Web Services account users. In addition, federated access might be provided through the SAML-based authentication available with Athena drivers. For more information, see Using workgroups to control query access and costs (p. 629), IAM policies for accessing workgroups (p. 632), and Enabling federated access to the Athena API (p. 571).

For more information, see Granting Lake Formation permissions in the AWS Lake Formation Developer Guide.

Amazon S3 permissions for Athena query results locations

The query results locations in Amazon S3 for Athena cannot be registered with Lake Formation. Lake Formation permissions do not limit access to these locations. Unless you limit access, Athena users can access query result files and metadata when they do not have Lake Formation permissions for the data. To avoid this, we recommend that you use workgroups to specify the location for query results and align workgroup membership with Lake Formation permissions. You can then use IAM permissions policies to limit access to query results locations. For more information about query results, see Working with query results, recent queries, and output files (p. 344).

Athena workgroup memberships to query history

Athena query history exposes a list of saved queries and complete query strings. Unless you use workgroups to separate access to query histories, Athena users who are not authorized to query data in Lake Formation are able to view query strings run on that data, including column names, selection
criteria, and so on. We recommend that you use workgroups to separate query histories, and align Athena workgroup membership with Lake Formation permissions to limit access. For more information, see Using workgroups to control query access and costs (p. 629).

Lake Formation permissions to data

In addition to the baseline permission to use Lake Formation, Athena users must have Lake Formation permissions to access resources that they query. These permissions are granted and managed by a Lake Formation administrator. For more information, see Security and access control to metadata and data in the AWS Lake Formation Developer Guide.

IAM permissions to write to Amazon S3 locations

Lake Formation permissions to Amazon S3 do not include the ability to write to Amazon S3. Create Table As Statements (CTAS) require write access to the Amazon S3 location of tables. To run CTAS queries on data registered with Lake Formation, Athena users must have IAM permissions to write to the table Amazon S3 locations in addition to the appropriate Lake Formation permissions to read the data locations. For more information, see Creating a table from query results (CTAS) (p. 281).

Permissions to encrypted data, metadata, and Athena query results

Underlying source data in Amazon S3 and metadata in the Data Catalog that is registered with Lake Formation can be encrypted. There is no change to the way that Athena handles encryption of query results when using Athena to query data registered with Lake Formation. For more information, see Encrypting Athena query results stored in Amazon S3 (p. 523).

- **Encrypting source data** – Encryption of Amazon S3 data locations source data is supported. Athena users who query encrypted Amazon S3 locations that are registered with Lake Formation need permissions to encrypt and decrypt data. For more information about requirements, see Supported Amazon S3 encryption options (p. 521) and Permissions to encrypted data in Amazon S3 (p. 522).

- **Encrypting metadata** – Encrypting metadata in the Data Catalog is supported. For principals using Athena, identity-based policies must allow the “kms:GenerateDataKey”, “kms:Decrypt”, and “kms:Encrypt” actions for the key used to encrypt metadata. For more information, see Encrypting your Data Catalog in the AWS Glue Developer Guide and Access from Athena to encrypted metadata in the AWS Glue Data Catalog (p. 553).

Resource-based permissions for Amazon S3 buckets in external accounts (optional)

To query an Amazon S3 data location in a different account, a resource-based IAM policy (bucket policy) must allow access to the location. For more information, see Cross-account access in Athena to Amazon S3 buckets (p. 537).

For information about accessing a Data Catalog in another account, see Athena cross-account Data Catalog access (p. 584).

Applying Lake Formation permissions to existing databases and tables

If you are new to Athena and you use Lake Formation to configure access to query data, you do not need to configure IAM policies so that users can read Amazon S3 data and create metadata. You can use Lake Formation to administer permissions.
Registering data with Lake Formation and updating IAM permissions policies is not a requirement. If data is not registered with Lake Formation, Athena users who have appropriate permissions in Amazon S3—and AWS Glue, if applicable—can continue to query data not registered with Lake Formation.

If you have existing Athena users who query data not registered with Lake Formation, you can update IAM permissions for Amazon S3—and the AWS Glue Data Catalog, if applicable—so that you can use Lake Formation permissions to manage user access centrally. For permission to read Amazon S3 data locations, you can update resource-based and identity-based policies to modify Amazon S3 permissions. For access to metadata, if you configured resource-level policies for fine-grained access control with AWS Glue, you can use Lake Formation permissions to manage access instead.

For more information, see Fine-grained access to databases and tables in the AWS Glue Data Catalog (p. 540) and Upgrading AWS Glue data permissions to the AWS Lake Formation model in the AWS Lake Formation Developer Guide.

Using Lake Formation and the Athena JDBC and ODBC drivers for federated access to Athena

The Athena JDBC and ODBC drivers support SAML 2.0-based federation with Athena using Okta and Microsoft Active Directory Federation Services (AD FS) identity providers. By integrating Amazon Athena with AWS Lake Formation, you enable SAML-based authentication to Athena with corporate credentials. With Lake Formation and AWS Identity and Access Management (IAM), you can maintain fine-grained, column-level access control over the data available to the SAML user. With the Athena JDBC and ODBC drivers, federated access is available for tool or programmatic access.

To use Athena to access a data source controlled by Lake Formation, you need to enable SAML 2.0-based federation by configuring your identity provider (IdP) and AWS Identity and Access Management (IAM) roles. For detailed steps, see Tutorial: Configuring federated access for Okta users to Athena using Lake Formation and JDBC (p. 590).

Prerequisites

To use Amazon Athena and Lake Formation for federated access, you must meet the following requirements:

- You manage your corporate identities using an existing SAML-based identity provider, such as Okta or Microsoft Active Directory Federation Services (AD FS).
- You use the AWS Glue Data Catalog as a metadata store.
- You define and manage permissions in Lake Formation to access databases, tables, and columns in AWS Glue Data Catalog. For more information, see the AWS Lake Formation Developer Guide.
- You use version 2.0.14 or later of the Athena JDBC driver or version 1.1.3 or later of the Athena ODBC driver (p. 185).

Considerations and limitations

When using the Athena JDBC or ODBC driver and Lake Formation to configure federated access to Athena, keep in mind the following points:

- Currently, the Athena JDBC driver and ODBC drivers support the Okta and Microsoft Active Directory Federation Services (AD FS) identity providers. Although the Athena JDBC driver has a generic SAML class that can be extended to use other identity providers, support for custom extensions that enable other identity providers (IdPs) for use with Athena may be limited.
- Currently, you cannot use the Athena console to configure support for IdP and SAML use with Athena. To configure this support, you use the third-party identity provider, the Lake Formation and IAM management consoles, and the JDBC or ODBC driver client.
• You should understand the SAML 2.0 specification and how it works with your identity provider before you configure your identity provider and SAML for use with Lake Formation and Athena.

• SAML providers and the Athena JDBC and ODBC drivers are provided by third parties, so support through AWS for issues related to their use may be limited.

Topics
• Tutorial: Configuring federated access for Okta users to Athena using Lake Formation and JDBC (p. 590)

Tutorial: Configuring federated access for Okta users to Athena using Lake Formation and JDBC

This tutorial shows you how to configure Okta, AWS Lake Formation, AWS Identity and Access Management permissions, and the Athena JDBC driver to enable SAML-based federated use of Athena. Lake Formation provides fine-grained access control over the data that is available in Athena to the SAML-based user. To set up this configuration, the tutorial uses the Okta developer console, the AWS IAM and Lake Formation consoles, and the SQL Workbench/J tool.

Prerequisites

This tutorial assumes that you have done the following:

• Created an Amazon Web Services account. To create an account, visit the Amazon Web Services home page.
• Set up a query results location (p. 344) for Athena in Amazon S3.
• Registered an Amazon S3 data bucket location with Lake Formation.
• Defined a database and tables on the AWS Glue Data Catalog that point to your data in Amazon S3.
  • If you have not yet defined a table, either run a AWS Glue crawler or use Athena to define a database and one or more tables (p. 251) for the data that you want to access.
• This tutorial uses a table based on the NYC taxi trips dataset available in the Registry of open data on AWS. The tutorial uses the database name tripdb and the table name nyctaxi.

Tutorial Steps

• Step 1: Create an Okta account (p. 590)
• Step 2: Add users and groups to Okta (p. 591)
• Step 3: Set up an Okta application for SAML authentication (p. 598)
• Step 4: Create an AWS SAML Identity Provider and Lake Formation access IAM role (p. 605)
• Step 5: Add the IAM role and SAML Identity Provider to the Okta application (p. 609)
• Step 6: Grant user and group permissions through AWS Lake Formation (p. 615)
• Step 7: Verify access through the Athena JDBC client (p. 619)
• Conclusion (p. 628)
• Related resources (p. 628)

Step 1: Create an Okta account

This tutorial uses Okta as a SAML-based identity provider. If you do not already have an Okta account, you can create a free one. An Okta account is required so that you can create an Okta application for SAML authentication.
To create an Okta account

1. To use Okta, navigate to the Okta developer sign up page and create a free Okta trial account. The Developer Edition Service is free of charge up to the limits specified by Okta at developer.okta.com/pricing.

2. When you receive the activation email, activate your account.

   An Okta domain name will be assigned to you. Save the domain name for reference. Later, you use the domain name (<okta-idp-domain>) in the JDBC string that connects to Athena.

Step 2: Add users and groups to Okta

In this step, you use the Okta console to perform the following tasks:

- Create two Okta users.
- Create two Okta groups.
- Add one Okta user to each Okta group.

To add users to Okta

1. After you activate your Okta account, log in as administrative user to the assigned Okta domain.

2. In the left navigation pane, choose Directory, and then choose People.

3. Choose Add Person to add a new user who will access Athena through the JDBC driver.

4. In the Add Person dialog box, enter the required information.

   - Enter values for First name and Last name. This tutorial uses athena-okta-user.
   - Enter a Username and Primary email. This tutorial uses athena-okta-user@anycompany.com.
   - For Password, choose Set by admin, and then provide a password. This tutorial clears the option for User must change password on first login; your security requirements may vary.
Add Person

User type: User

First name: athena-okta-user

Last name: athena-okta-user

Username: athena-okta-user@anycompany.com

Primary email: athena-okta-user@anycompany.com

Secondary email (optional): 

Groups (optional): 

Password: Set by admin

User must change password on first login

Save | Save and Add Another | Cancel
5. Choose **Save and Add Another**.

6. Enter the information for another user. This example adds the business analyst user **athena-ba-user@anycompany.com**.
7. Choose Save.
In the following procedure, you provide access for two Okta groups through the Athena JDBC driver by adding a "Business Analysts" group and a "Developer" group.

**To add Okta groups**

1. In the Okta navigation pane, choose **Directory**, and then choose **Groups**.
2. On the **Groups** page, choose **Add Group**.

3. In the **Add Group** dialog box, enter the required information.
   - For **Name**, enter **lf-business-analyst**.
   - For **Group Description**, enter **Business Analysts**.

4. Choose **Add Group**.
5. On the Groups page, choose Add Group again. This time you will enter information for the Developer group.

6. Enter the required information.
   - For Name, enter lf-developer.
   - For Group Description, enter Developers.

7. Choose Add Group.

Now that you have two users and two groups, you are ready to add a user to each group.

**To add users to groups**

1. On the Groups page, choose the lf-developer group that you just created. You will add one of the Okta users that you created as a developer to this group.

   ![Groups page](image)

   2. Choose Manage People.

   ![Manage People](image)
3. From the **Not Members** list, choose **athena-okta-user**.

![User management interface](image)

The entry for the user moves from the **Not Members** list on the left to the **Members** list on the right.
4. Choose **Save**.

5. Choose **Back to Group**, or choose **Directory**, and then choose **Groups**.

6. Choose the **lf-business-analyst** group.

7. Choose **Manage People**.

8. Add the **athena-ba-user** to the **Members** list of the **lf-business-analyst** group, and then choose **Save**.

9. Choose **Back to Group**, or choose **Directory, Groups**.

   The **Groups** page now shows that each group has one Okta user.

---

**Step 3: Set up an Okta application for SAML authentication**

In this step, you use the Okta developer console to perform the following tasks:

- Add a SAML application for use with AWS.
- Assign the application to the Okta user.
- Assign the application to an Okta group.
• Download the resulting identity provider metadata for later use with AWS.

To add an application for SAML authentication

1. In the Okta navigation pane, choose Applications, Applications so that you can configure an Okta application for SAML authentication to Athena.

2. Click Browse App Catalog.

3. In the search box, enter Redshift.

4. Choose Amazon Web Services Redshift. The Okta application in this tutorial uses the existing SAML integration for Amazon Redshift.

5. On the Amazon Web Services Redshift page, choose Add to create a SAML-based application for Amazon Redshift.

6. For Application label, enter Athena-LakeFormation-Okta, and then choose Done.
Now that you have created an Okta application, you can assign it to the users and groups that you created.

**To assign the application to users and groups**

1. On the **Applications** page, choose the **Athena-LakeFormation-Okta** application.
2. On the **Assignments** tab, choose **Assign, Assign to People**.
3. In the **Assign Athena-LakeFormation-Okta to People** dialog box, find the **athena-okta-user** user that you created previously.

4. Choose **Assign** to assign the user to the application.

5. Choose **Save and Go Back**.

6. Choose **Done**.

7. On the **Assignments** tab for the **Athena-LakeFormation-Okta** application, choose **Assign, Assign to Groups**.

8. For **lf-business-analyst**, choose **Assign** to assign the **Athena-LakeFormation-Okta** application to the **lf-business-analyst** group, and then choose **Done**.
The group appears in the list of groups for the application.
Now you are ready to download the identity provider application metadata for use with AWS.

**To download the application metadata**

1. Choose the Okta application **Sign On** tab, and then right-click **Identity Provider metadata**.
2. Choose **Save Link As** to save the identity provider metadata, which is in XML format, to a file. Give it a name that you recognize (for example, `Athena-LakeFormation-idp-metadata.xml`).
Step 4: Create an AWS SAML Identity Provider and Lake Formation access IAM role

In this step, you use the AWS Identity and Access Management (IAM) console to perform the following tasks:

- Create an identity provider for AWS.
- Create an IAM role for Lake Formation access.
- Add the AmazonAthenaFullAccess managed policy to the role.
- Add a policy for Lake Formation and AWS Glue to the role.
- Add a policy for Athena query results to the role.

To create an AWS SAML identity provider

1. Sign in to the Amazon Web Services account console as Amazon Web Services account administrator and navigate to the IAM console (https://console.aws.amazon.com/iam/).
2. In the navigation pane, choose Identity providers, and then click Add provider.
3. On the Configure provider screen, enter the following information:
   - For Provider type, choose SAML.
   - For Provider name, enter AthenaLakeFormationOkta.
   - For Metadata document, use the Choose file option to upload the identity provider (IdP) metadata XML file that you downloaded.
4. Choose Add provider.

Next, you create an IAM role for AWS Lake Formation access. You add two inline policies to the role. One policy provides permissions to access Lake Formation and the AWS Glue APIs. The other policy provides access to Athena and the Athena query results location in Amazon S3.

To create an IAM role for AWS Lake Formation access

1. In the IAM console navigation pane, choose Roles, and then choose Create role.
2. On the Create role page, perform the following steps:
a. For **Select type of trusted entity**, choose **SAML 2.0 Federation**.

b. For **SAML provider**, select **AthenaLakeFormationOkta**.

c. For **SAML provider**, select the option **Allow programmatic and AWS Management Console access**.

d. Choose **Next: Permissions**.

3. On the **Attach Permissions policies** page, for **Filter policies**, enter **Athena**.

4. Select the **AmazonAthenaFullAccess** managed policy, and then choose **Next: Tags**.
5. On the **Add tags** page, choose **Next: Review**.
6. On the **Review** page, for **Role name**, enter a name for the role (for example, *Athena-LakeFormation-OktaRole*), and then choose **Create role**.
Next, you add inline policies that allow access to Lake Formation, AWS Glue APIs, and Athena query results in Amazon S3.

Whenever you use IAM policies, make sure that you follow IAM best practices. For more information, see Security best practices in IAM in the IAM User Guide.

To add an inline policy to the role for Lake Formation and AWS Glue

1. From the list of roles in the IAM console, choose the newly created Athena-LakeFormation-OktaRole.
2. On the Summary page for the role, on the Permissions tab, choose Add inline policy.
3. On the Create policy page, choose JSON.
4. Add an inline policy like the following that provides access to Lake Formation and the AWS Glue APIs.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "lakeformation:GetDataAccess",
            "glue:GetTable",
            "glue:GetTables",
            "glue:GetDatabase",
            "glue:GetDatabases",
            "glue:CreateDatabase",
            "glue:GetUserDefinedFunction",
            "glue:GetUserDefinedFunctions"
         ],
         "Resource": "*"
      }
   ]
}
```
5. Choose Review policy.
6. For Name, enter a name for the policy (for example, LakeFormationGlueInlinePolicy).
7. Choose Create policy.

To add an inline policy to the role for the Athena query results location

1. On the Summary page for the Athena-LakeFormation-OktaRole role, on the Permissions tab, choose Add inline policy.
2. On the Create policy page, choose JSON.
3. Add an inline policy like the following that allows the role access to the Athena query results location. Replace the <athena-query-results-bucket> placeholders in the example with the name of your Amazon S3 bucket.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "AthenaQueryResultsPermissionsForS3",
         "Effect": "Allow",
         "Action": [
            "s3:ListBucket",
            "s3:PutObject",
            "s3:GetObject"
         ]
      }
   ]
}
```
5. For Name, enter a name for the policy (for example, AthenaQueryResultsInlinePolicy).
6. Choose Create policy.

Next, you copy the ARN of the Lake Formation access role and the ARN of the SAML provider that you created. These are required when you configure the Okta SAML application in the next section of the tutorial.

To copy the role ARN and SAML identity provider ARN

1. In the IAM console, on the Summary page for the Athena-LakeFormation-OktaRole role, choose the Copy to clipboard icon next to Role ARN. The ARN has the following format:
   
   ```
   arn:aws:iam::<account-id>:role/Athena-LakeFormation-OktaRole
   ```

2. Save the full ARN securely for later reference.
3. In the IAM console navigation pane, choose Identity providers.
4. Choose the AthenaLakeFormationOkta provider.
5. On the Summary page, choose the Copy to clipboard icon next to Provider ARN. The ARN should look like the following:
   
   ```
   arn:aws:iam::<account-id>:saml-provider/AthenaLakeFormationOkta
   ```

6. Save the full ARN securely for later reference.

Step 5: Add the IAM role and SAML Identity Provider to the Okta application

In this step, you return to the Okta developer console and perform the following tasks:

- Add user and group Lake Formation URL attributes to the Okta application.
- Add the ARN for the identity provider and the ARN for the IAM role to the Okta application.
- Copy the Okta application ID. The Okta application ID is required in the JDBC profile that connects to Athena.

To add user and group Lake Formation URL attributes to the Okta application

1. Sign into the Okta developer console.
2. Choose the Applications tab, and then choose the Athena-LakeFormation-Okta application.
3. Choose on the Sign On tab for the application, and then choose Edit.
Choose Attributes (optional) to expand it.
5. For **Attribute Statements (optional)**, add the following attribute:

   - For **Name**, enter `https://lakeformation.amazon.com/SAML/Attributes/Username`.
6. Under **Group Attribute Statements (optional)**, add the following attribute:
   - For **Name**, enter `https://lakeformation.amazon.com/SAML/Attributes/Groups`.
   - For **Name format**, enter `Basic`.
   - For **Filter**, choose `Matches regex`, and then enter `.*` in the filter box.

7. Scroll down to the **Advanced Sign-On Settings** section, where you will add the identity provider and IAM Role ARNs to the Okta application.
To add the ARNs for the identity provider and IAM role to the Okta application

1. For **Idp ARN and Role ARN**, enter the AWS identity provider ARN and role ARN as comma separated values in the format `<saml-arn>,<role-arn>`. The combined string should look like the following:

```
arn:aws:iam::<account-id>:saml-provider/AthenaLakeFormationOkta,arn:aws:iam::<account-id>:role/Athena-LakeFormation-OktaRole
```

2. Choose **Save**.

Next, you copy the Okta application ID. You will require this later for the JDBC string that connects to Athena.

**To find and copy the Okta application ID**

1. Choose the **General** tab of the Okta application.
2. Scroll down to the **App Embed Link** section.

3. From **Embed Link**, copy and securely save the Okta application ID portion of the URL. The Okta application ID is the part of the URL after `amazon_aws_redshift/` but before the next forward slash. For example, if the URL contains `amazon_aws_redshift/aaa/bbb`, the application ID is `aaa`.

![Athena-LakeFormation-Okta](image)

**App Embed Link**

**Embed Link**

You can use the URL below to sign into Amazon Web Services Redshift from a portal or other location outside of Okta.

```
https://dev-_____.okta.com/home/amazon_aws_redshift
```

**Application Login Page**
Note
The embed link cannot be used to log directly into the Athena console to view databases. The Lake Formation permissions for SAML users and groups are recognized only when you use the JDBC or ODBC driver to submit queries to Athena. To view the databases, you can use the SQL Workbench/J tool, which uses the JDBC driver to connect to Athena. The SQL Workbench/J tool is covered in Step 7: Verify access through the Athena JDBC client (p. 619).

Step 6: Grant user and group permissions through AWS Lake Formation

In this step, you use the Lake Formation console to grant permissions on a table to the SAML user and group. You perform the following tasks:

- Specify the ARN of the Okta SAML user and associated user permissions on the table.
- Specify the ARN of the Okta SAML group and associated group permissions on the table.
- Verify the permissions that you granted.

To grant permissions in Lake Formation for the Okta user

1. Sign in as data lake administrator to the AWS Management Console.
3. From the navigation pane, choose Tables, and then select the table that you want to grant permissions for. This tutorial uses the nyctaxi table from the tripdb database.
4. From Actions, choose Grant.
5. In the **Grant permissions** dialog, enter the following information:

a. Under **SAML and Amazon QuickSight users and groups**, enter the Okta SAML user ARN in the following format:

```
arn:aws:iam::<account-id>:saml-provider/AthenaLakeFormationOkta:user/<athena-okta-user>@<anycompany.com>
```

b. For **Columns**, for **Choose filter type**, and optionally choose **Include columns** or **Exclude columns**.

c. Use the **Choose one or more columns** dropdown under the filter to specify the columns that you want to include or exclude for or from the user.

d. For **Table permissions**, choose **Select**. This tutorial grants only the SELECT permission; your requirements may vary.
6. Choose Grant.

Now you perform similar steps for the Okta group.

To grant permissions in Lake Formation for the Okta group

1. On the Tables page of the Lake Formation console, make sure that the nyctaxi table is still selected.
2. From Actions, choose Grant.
3. In the Grant permissions dialog, enter the following information:
   a. Under SAML and Amazon QuickSight users and groups, enter the Okta SAML group ARN in the following format:
      ```
      arn:aws:iam::<account-id>:saml-provider/AthenaLakeFormationOkta:group/lf-business-analyst
      ```
   b. For Columns, Choose filter type, choose Include columns.
   c. For Choose one or more columns, choose the first three columns of the table.
   d. For Table permissions, choose the specific access permissions to grant. This tutorial grants only the SELECT permission; your requirements may vary.
4. Choose **Grant**.

5. To verify the permissions that you granted, choose **Actions, View permissions**.
Step 7: Verify access through the Athena JDBC client

Now you are ready to use a JDBC client to perform a test connection to Athena as the Okta SAML user.

In this section, you perform the following tasks:

- Prepare the test client – Download the Athena JDBC driver, install SQL Workbench, and add the driver to Workbench. This tutorial uses SQL Workbench to access Athena through Okta authentication and to verify Lake Formation permissions.
- In SQL Workbench:
  - Create a connection for the Athena Okta user.
  - Run test queries as the Athena Okta user.
• Create and test a connection for the business analyst user.
• In the Okta console, add the business analyst user to the developer group.
• In the Lake Formation console, configure table permissions for the developer group.
• In SQL Workbench, run test queries as the business analyst user and verify how the change in permissions affects the results.

**To prepare the test client**

1. Download and extract the Lake Formation compatible Athena JDBC driver (2.0.14 or later version) from [Using Athena with the JDBC driver](p. 183).
2. Download and install the free [SQL Workbench/J](p. 183) SQL query tool, available under a modified Apache 2.0 license.
3. In SQL Workbench, choose **File**, and then choose **Manage Drivers**.

   ![SQL Workbench/J interface](image)

4. In the **Manage Drivers** dialog box, perform the following steps:
   a. Choose the new driver icon.
b. For **Name**, enter *Athena*.

c. For **Library**, browse to and choose the Simba Athena JDBC .jar file that you just downloaded.

d. Choose **OK**.

You are now ready to create and test a connection for the Athena Okta user.

**To create a connection for the Okta user**

1. Choose **File**, **Connect window**.

2. In the **Connection profile** dialog box, create a connection by entering the following information:

   - In the name box, enter **Athena_Okta_User_Connection**.
   - For **Driver**, choose the Simba Athena JDBC Driver.
   - For **URL**, do one of the following:
     - To use a connection URL, enter a single-line connection string. The following example adds line breaks for readability.

```
jdbc:awsathena://AwsRegion=region-id;
S3OutputLocation=s3://athena-query-results-bucket/athena_results;
AwsCredentialsProviderClass=com.simba.athena.iamsupport.plugin.OktaCredentialsProvider;
```
To use an AWS profile-based URL, perform the following steps:

1. Configure an AWS profile that has an AWS credentials file like the following example.

```
[athena_lf_dev]
plugin_name=com.simba.athena.iamsupport.plugin.OktaCredentialsProvider
idp_host=okta-idp-domain
app_id=okta-app-id
uid=athena-okta-user@anycompany.com
pwd=password
```

2. For URL, enter a single-line connection string like the following example. The example adds line breaks for readability.

```
jdbc:awsathena://AwsRegion=region-id;
S3OutputLocation=s3://athena-query-results-bucket/athena_results;
profile=athena_lf_dev;
SSL_Insecure=true;
LakeFormationEnabled=true;
```

Note that these examples are basic representations of the URL needed to connect to Athena. For the full list of parameters supported in the URL, refer to the Simba Athena JDBC driver installation guide (p. 185). The JDBC installation guide also provides sample Java code for connecting to Athena programmatically.

The following image shows a SQL Workbench connection profile that uses a connection URL.

Now that you have established a connection for the Okta user, you can test it by retrieving some data.
To test the connection for the Okta user

1. Choose Test, and then verify that the connection succeeds.
2. From the SQL Workbench **Statement** window, run the following SQL DESCRIBE command. Verify that all columns are displayed.

   ```sql
   DESCRIBE "tripdb"."nyctaxi"
   ```

   ![SQL Workbench screenshot showing DESCRIBE command output]

3. From the SQL Workbench **Statement** window, run the following SQL SELECT command. Verify that all columns are displayed.

   ```sql
   SELECT * FROM tripdb.nyctaxi LIMIT 5
   ```
Next, you verify that the **athena-ba-user**, as a member of the **lf-business-analyst** group, has access to only the first three columns of the table that you specified earlier in Lake Formation.

**To verify access for the athena-ba-user**

1. In SQL Workbench, in the **Connection profile** dialog box, create another connection profile.
   - For the connection profile name, enter **Athena_Okta_Group_Connection**.
   - For **Driver**, choose the Simba Athena JDBC driver.
   - For **URL**, do one of the following:
     - To use a connection URL, enter a single-line connection string. The following example adds line breaks for readability.

```
jdbc:awsathena://AwsRegion=region-id;
S3OutputLocation=s3://athena-query-results-bucket/athena_results;
AwsCredentialsProviderClass=com.simba.athena.iamsupport.plugin.OktaCredentialsProvider;
user=athena-ba-user@anycompany.com;
pASSWORD=password;
idp_host=okta-idp-domain;
App_ID=okta-application-id;
SSL_Insecure=true;
LakeFormationEnabled=true;
```

     - To use an AWS profile-based URL, perform the following steps:

   1. Configure an AWS profile that has a credentials file like the following example.

```
[athena_lf_ba]
plugin_name=com.simba.athena.iamsupport.plugin.OktaCredentialsProvider
idp_host=okta-idp-domain
app_id=okta-application-id
uid=athena-ba-user@anycompany.com
pwd=password
```

   2. For **URL**, enter a single-line connection string like the following. The example adds line breaks for readability.
2. Choose **Test** to confirm that the connection is successful.

3. From the **SQL Statement** window, run the same DESCRIBE and SELECT SQL commands that you did before and examine the results.

Because **athena-ba-user** is a member of the **lf-business-analyst** group, only the first three columns that you specified in the Lake Formation console are returned.

![SQL Workbench/J Athena_AD_Group_Connection - Default.wks](image1)

```sql
1describe tripdb.nyctaxi
```

<table>
<thead>
<tr>
<th>COLUMN_NAME</th>
<th>DATA_TYPE</th>
<th>PK</th>
<th>NULLABLE</th>
<th>DEFAULT</th>
<th>AUTOINCREMENT</th>
<th>COMPUTED</th>
<th>REMARKS</th>
<th>POSITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>vendor</td>
<td>string</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>1</td>
</tr>
<tr>
<td>peer_pickup_datetime</td>
<td>string(255)</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>2</td>
</tr>
<tr>
<td>peer_dropoff_datetime</td>
<td>string(255)</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>3</td>
</tr>
</tbody>
</table>

```sql
1select * from tripdb.nyctaxi limit 5
```

<table>
<thead>
<tr>
<th>vendor</th>
<th>peer_pickup_datetime</th>
<th>peer_dropoff_datetime</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2017-01-01 00:01:15</td>
<td>2017-01-01 00:11:05</td>
</tr>
<tr>
<td></td>
<td>2017-01-01 00:03:34</td>
<td>2017-01-01 00:09:00</td>
</tr>
<tr>
<td></td>
<td>2017-01-01 00:04:02</td>
<td>2017-01-01 00:12:55</td>
</tr>
<tr>
<td></td>
<td>2017-01-01 00:01:40</td>
<td>2017-01-01 00:14:23</td>
</tr>
<tr>
<td></td>
<td>2017-01-01 00:00:51</td>
<td>2017-01-01 00:18:55</td>
</tr>
</tbody>
</table>

Next, you return to the Okta console to add the **athena-ba-user** to the **lf-developer** Okta group.

**To add the athena-ba-user to the lf-developer group**

1. Sign in to the Okta console as an administrative user of the assigned Okta domain.
2. Choose **Directory**, and then choose **Groups**.
3. On the Groups page, choose the **lf-developer** group.

![Groups](image)

4. Choose **Manage People**.
5. From the **Not Members** list, choose the **athena-ba-user** to add it to the **lf-developer** group.
6. Choose **Save**.

Now you return to the Lake Formation console to configure table permissions for the **lf-developer** group.

**To configure table permissions for the lf-developer-group**

1. Log into the Lake Formation console as Data Lake administrator.
2. In the navigation pane, choose **Tables**.
3. Select the **nyctaxi** table.
4. Choose Actions, Grant.
5. In the Grant Permissions dialog, enter the following information:
   • For SAML and Amazon QuickSight users and groups, enter the Okta SAML lf-developer group ARN in the following format:
   • For Columns, Choose filter type, choose Include columns.
   • Choose the trip_type column.
   • For Table permissions, choose SELECT.
6. Choose Grant.

Now you can use SQL Workbench to verify the change in permissions for the lf-developer group. The change should be reflected in the data available to athena-ba-user, who is now a member of the lf-developer group.

To verify the change in permissions for athena-ba-user
1. Close the SQL Workbench program, and then re-open it.
2. Connect to the profile for athena-ba-user.
3. From the Statement window, issue the same SQL statements that you ran previously:

   This time, the trip_type column is displayed.

   Because athena-ba-user is now a member of both the lf-developer and lf-business-analyst groups, the combination of Lake Formation permissions for those groups determines the columns that are returned.
Conclusion

In this tutorial you configured Athena integration with AWS Lake Formation using Okta as the SAML provider. You used Lake Formation and IAM to control the resources that are available to the SAML user in your data lake AWS Glue Data Catalog.

Related resources

For related information, see the following resources.

- Using Athena with the JDBC driver (p. 183)
- Enabling federated access to the Athena API (p. 571)
- AWS Lake Formation Developer Guide
- Granting and revoking Data Catalog permissions in the AWS Lake Formation Developer Guide.
- Identity providers and federation in the IAM User Guide.
- Creating IAM SAML identity providers in the IAM User Guide.
Using workgroups to control query access and costs

Use workgroups to separate users, teams, applications, or workloads, to set limits on amount of data each query or the entire workgroup can process, and to track costs. Because workgroups act as resources, you can use resource-level identity-based policies to control access to a specific workgroup. You can also view query-related metrics in Amazon CloudWatch, control costs by configuring limits on the amount of data scanned, create thresholds, and trigger actions, such as Amazon SNS, when these thresholds are breached.

Workgroups integrate with IAM, CloudWatch, Amazon Simple Notification Service, and AWS Cost and Usage Reports as follows:

- IAM identity-based policies with resource-level permissions control who can run queries in a workgroup.
- Athena publishes the workgroup query metrics to CloudWatch, if you enable query metrics.
- In Amazon SNS, you can create Amazon SNS topics that issue alarms to specified workgroup users when data usage controls for queries in a workgroup exceed your established thresholds.
- When you tag a workgroup with a tag configured as a cost allocation tag in the Billing and Cost Management console, the costs associated with running queries in that workgroup appear in your Cost and Usage Reports with that cost allocation tag.

Topics
- Using workgroups for running queries (p. 629)
- Controlling costs and monitoring queries with CloudWatch metrics and events (p. 647)

See also the AWS Big Data Blog post Separate queries and managing costs using Amazon Athena workgroups, which shows you how to use workgroups to separate workloads, control user access, and manage query usage and costs.

Using workgroups for running queries

We recommend using workgroups to isolate queries for teams, applications, or different workloads. For example, you may create separate workgroups for two different teams in your organization. You can also separate workloads. For example, you can create two independent workgroups, one for automated scheduled applications, such as report generation, and another for ad-hoc usage by analysts. You can switch between workgroups.

Topics
- Benefits of using workgroups (p. 630)
- How workgroups work (p. 630)
- Setting up workgroups (p. 631)
- IAM policies for accessing workgroups (p. 632)
- Workgroup settings (p. 638)
- Managing workgroups (p. 639)
Benefits of using workgroups

Workgroups allow you to:

<table>
<thead>
<tr>
<th>Isolate users, teams, applications, or workloads into groups.</th>
<th>Each workgroup has its own distinct query history and a list of saved queries. For more information, see How workgroups work (p. 630). For all queries in the workgroup, you can choose to configure workgroup settings. They include an Amazon S3 location for storing query results, expected bucket owner, encryption, and control of objects written to the query results bucket. You can also enforce workgroup settings. For more information, see Workgroup settings (p. 638).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enforce costs constraints.</td>
<td>You can set two types of cost constraints for queries in a workgroup:</td>
</tr>
</tbody>
</table>
| • Per-query limit is a threshold for the amount of data scanned for each query. Athena cancels queries when they exceed the specified threshold. The limit applies to each running query within a workgroup. You can set only one per-query limit and update it if needed.  
• Per-workgroup limit is a threshold you can set for each workgroup for the amount of data scanned by queries in the workgroup. Breaching a threshold activates an Amazon SNS alarm that triggers an action of your choice, such as sending an email to a specified user. You can set multiple per-workgroup limits for each workgroup. | For detailed steps, see Setting data usage control limits (p. 655). |
| Track query-related metrics for all workgroup queries in CloudWatch. | For each query that runs in a workgroup, if you configure the workgroup to publish metrics, Athena publishes them to CloudWatch. You can view query metrics (p. 648) for each of your workgroups within the Athena console. In CloudWatch, you can create custom dashboards, and set thresholds and alarms on these metrics. |

How workgroups work

Workgroups in Athena have the following characteristics:

• By default, each account has a primary workgroup and the default permissions allow all authenticated users access to this workgroup. The primary workgroup cannot be deleted.

• Each workgroup that you create shows saved queries and query history only for queries that ran in it, and not for all queries in the account. This separates your queries from other queries within an account and makes it more efficient for you to locate your own saved queries and queries in history.

• Disabling a workgroup prevents queries from running in it, until you enable it. Queries sent to a disabled workgroup fail, until you enable it again.

• If you have permissions, you can delete an empty workgroup, and a workgroup that contains saved queries. In this case, before deleting a workgroup, Athena warns you that saved queries are deleted. Before deleting a workgroup to which other users have access, make sure its users have access to other workgroups in which they can continue to run queries.
• You can set up workgroup-wide settings and enforce their usage by all queries that run in a workgroup. The settings include query results location in Amazon S3, expected bucket owner, encryption, and control of objects written to the query results bucket.

  **Important**
  When you enforce workgroup-wide settings, all queries that run in this workgroup use workgroup settings. This happens even if their client-side settings may differ from workgroup settings. For information, see Workgroup settings override client-side settings (p. 638).

**Limitations for workgroups**

• You can create up to 1000 workgroups per Region in your account.
• The primary workgroup cannot be deleted.
• You can open up to ten query tabs within each workgroup. When you switch between workgroups, your query tabs remain open for up to three workgroups.

**Setting up workgroups**

Setting up workgroups involves creating them and establishing permissions for their usage. First, decide which workgroups your organization needs, and create them. Next, set up IAM workgroup policies that control user access and actions on a workgroup resource. Users with access to these workgroups can now run queries in them.

  **Note**
  Use these tasks for setting up workgroups when you begin to use them for the first time. If your Athena account already uses workgroups, each account’s user requires permissions to run queries in one or more workgroups in the account. Before you run queries, check your IAM policy to see which workgroups you can access, adjust your policy if needed, and switch (p. 643) to a workgroup you intend to use.

By default, if you have not created any workgroups, all queries in your account run in the primary workgroup.

Athena displays the current workgroup in the **Workgroup** option on the upper right of the console. You can use this option to switch workgroups. When you run queries, they run in the current workgroup. You can run queries in the context of a workgroup in the console, through API operations, through the command line interface, or through a client application by using the JDBC or ODBC driver. When you have access to a workgroup, you can view the workgroup’s settings, metrics, and data usage control limits. With additional permissions, you can edit the settings and data usage control limits.

**To set up workgroups**

1. Decide which workgroups to create. For example, you can decide the following:

   • Who can run queries in each workgroup, and who owns workgroup configuration. This determines IAM policies you create. For more information, see IAM policies for accessing workgroups (p. 632).
   • Which locations in Amazon S3 to use for the query results for queries that run in each workgroup. A location must exist in Amazon S3 before you can specify it for the workgroup query results. All users who use a workgroup must have access to this location. For more information, see Workgroup settings (p. 638).
   • Whether the owner of the Amazon S3 query results bucket has full control over new objects that are written to the bucket. For example, if your query result location is owned by another account, you can grant ownership and full control over your query results to the other account. For more information, see AclConfiguration.
• Specify the ID of the AWS account that you expect to be the owner of the output location bucket. This is an optional added security measure. If the account ID of the bucket owner does not match the ID that you specify here, attempts to output to the bucket will fail. For more information, see Verifying bucket ownership with bucket owner condition in the Amazon S3 User Guide. This setting does not apply to CTAS, INSERT INTO, or UNLOAD statements.

• Which encryption settings are required, and which workgroups have queries that must be encrypted. We recommend that you create separate workgroups for encrypted and non-encrypted queries. That way, you can enforce encryption for a workgroup that applies to all queries that run in it. For more information, see Encrypting Athena query results stored in Amazon S3 (p. 523).

2. Create workgroups as needed, and add tags to them. For steps, see Create a workgroup (p. 640).

3. Create IAM policies for your users, groups, or roles to enable their access to workgroups. The policies establish the workgroup membership and access to actions on a workgroup resource. For detailed steps, see IAM policies for accessing workgroups (p. 632). For example JSON policies, see Access to workgroups and tags (p. 553).

4. Set workgroup settings. Specify a location in Amazon S3 for query results and optionally specify the expected bucket owner, encryption settings, and control of objects written to the query results bucket. You can enforce workgroup settings. For more information, see workgroup settings (p. 638).

   **Important**
   If you override client-side settings (p. 638), Athena will use the workgroup's settings. This affects queries that you run in the console, by using the drivers, the command line interface, or the API operations.
   While queries continue to run, automation built based on availability of results in a certain Amazon S3 bucket may break. We recommend that you inform your users before overriding. After workgroup settings are set to override, you can omit specifying client-side settings in the drivers or the API.

5. Notify users which workgroups to use for running queries. Send an email to inform your account's users about workgroup names that they can use, the required IAM policies, and the workgroup settings.

6. Configure cost control limits, also known as data usage control limits, for queries and workgroups. To notify you when a threshold is breached, create an Amazon SNS topic and configure subscriptions. For detailed steps, see Setting data usage control limits (p. 655) and Getting started with Amazon SNS in the Amazon Simple Notification Service Developer Guide.

7. Switch to the workgroup so that you can run queries. To run queries, switch to the appropriate workgroup. For detailed steps, see the section called “Specify a workgroup in which to run queries” (p. 644).

---

### IAM policies for accessing workgroups

To control access to workgroups, use resource-level IAM permissions or identity-based IAM policies. Whenever you use IAM policies, make sure that you follow IAM best practices. For more information, see Security best practices in IAM in the IAM User Guide.

The following procedure is specific to Athena.

For IAM-specific information, see the links listed at the end of this section. For information about example JSON workgroup policies, see Workgroup example policies (p. 633).

**To use the visual editor in the IAM console to create a workgroup policy**

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane on the left, choose Policies, and then choose Create policy.
3. **On the Visual editor** tab, choose **Choose a service**. Then choose Athena to add to the policy.

4. **Choose Select actions**, and then choose the actions to add to the policy. The visual editor shows the actions available in Athena. For more information, see Actions, resources, and condition keys for Amazon Athena in the Service Authorization Reference.

5. **Choose add actions** to type a specific action or use wildcards (*) to specify multiple actions.

   By default, the policy that you are creating allows the actions that you choose. If you chose one or more actions that support resource-level permissions to the workgroup resource in Athena, then the editor lists the workgroup resource.

6. **Choose Resources** to specify the specific workgroups for your policy. For example JSON workgroup policies, see Workgroup example policies (p. 633).

7. **Specify the workgroup resource as follows:**

   ```
   arn:aws:athena:<region>:<user-account>:workgroup/<workgroup-name>
   ```

8. **Choose Review policy**, and then type a Name and a Description (optional) for the policy that you are creating. Review the policy summary to make sure that you granted the intended permissions.

9. **Choose Create policy** to save your new policy.

10. **Attach this identity-based policy to a user, a group, or role.**

For more information, see the following topics in the Service Authorization Reference and IAM User Guide:

- Actions, resources, and condition keys for Amazon Athena
- Creating policies with the visual editor
- Adding and removing IAM policies
- Controlling access to resources

For example JSON workgroup policies, see Workgroup example policies (p. 633).

For a complete list of Amazon Athena actions, see the API action names in the Amazon Athena API Reference.

**Workgroup example policies**

This section includes example policies you can use to enable various actions on workgroups. Whenever you use IAM policies, make sure that you follow IAM best practices. For more information, see Security best practices in IAM in the IAM User Guide.

A workgroup is an IAM resource managed by Athena. Therefore, if your workgroup policy uses actions that take workgroup as an input, you must specify the workgroup's ARN as follows:

```
```

Where `<workgroup-name>` is the name of your workgroup. For example, for workgroup named `test_workgroup`, specify it as a resource as follows:

```
"Resource": ["arn:aws:athena:us-east-1:123456789012:workgroup/test_workgroup"]
```

For a complete list of Amazon Athena actions, see the API action names in the Amazon Athena API Reference. For more information about IAM policies, see Creating policies with the visual editor in the IAM User Guide. For more information about creating IAM policies for workgroups, see IAM policies for accessing workgroups (p. 632).

- Example Policy for Full Access to All Workgroups (p. 634)
IAM policies for accessing workgroups

- Example Policy for Full Access to a Specified Workgroup (p. 634)
- Example Policy for Running Queries in a Specified Workgroup (p. 635)
- Example Policy for Running Queries in the Primary Workgroup (p. 636)
- Example Policy for Management Operations on a Specified Workgroup (p. 636)
- Example Policy for Listing Workgroups (p. 637)
- Example Policy for Running and Stopping Queries in a Specific Workgroup (p. 637)
- Example Policy for Working with Named Queries in a Specific Workgroup (p. 637)

Example policy for full access to all workgroups

The following policy allows full access to all workgroup resources that might exist in the account. We recommend that you use this policy for those users in your account that must administer and manage workgroups for all other users.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "athena:*"
      ],
      "Resource": [ "*" ]
    }
  ]
}
```

Example policy for full access to a specified workgroup

The following policy allows full access to the single specific workgroup resource, named `workgroupA`. You could use this policy for users with full control over a particular workgroup.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "athena:ListEngineVersions",
        "athena:ListWorkGroups",
        "athena:ListDataCatalogs",
        "athena:ListDatabases",
        "athena:GetDatabase",
        "athena:ListTableMetadata",
        "athena:GetTableMetadata"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "athena:BatchGetQueryExecution",
        "athena:GetQueryExecution",
        "athena:ListQueryExecutions",
        "athena:StartQueryExecution",
        "athena:StopQueryExecution",
        "athena:GetQueryResults"
      ]
    }
  ]
}
```
IAM policies for accessing workgroups

Example Example policy for running queries in a specified workgroup

In the following policy, a user is allowed to run queries in the specified `workgroupA`, and view them. The user is not allowed to perform management tasks for the workgroup itself, such as updating or deleting it.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "athena:ListEngineVersions",
                "athena:ListWorkGroups",
                "athena:ListDataCatalogs",
                "athena:ListDatabases",
                "athena:GetDatabase",
                "athena:ListTableMetadata",
                "athena:GetTableMetadata"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "athena:GetWorkGroup",
                "athena:BatchGetQueryExecution",
                "athena:GetQueryExecution",
                "athena:ListQueryExecutions",
                "athena:StartQueryExecution",
                "athena:StopQueryExecution",
                "athena:GetQueryResults",
                "athena:GetQueryResultsStream",
                "athena:DeleteWorkGroup",
                "athena:UpdateWorkGroup",
                "athena:GetWorkGroup",
                "athena:CreateWorkGroup"
            ],
        }
    ]
}
```
IAM policies for accessing workgroups

Example policy for running queries in the primary workgroup

You can modify the preceding example to allow a particular user to also run queries in the primary workgroup.

**Note**

We recommend that you add the primary workgroup resource for all users who are otherwise configured to run queries in their designated workgroups. Adding this resource to their workgroup user policies is useful in case their designated workgroup is deleted or is disabled. In this case, they can continue running queries in the primary workgroup.

To allow users in your account to run queries in the primary workgroup, add a line that contains the ARN of the primary workgroup to the resource section of the Example Policy for Running Queries in a Specified Workgroup (p. 635), as in the following example.

```
arn:aws:athena:us-east-1:123456789012:workgroup/primary
```

Example policy for management operations on a specified workgroup

In the following policy, a user is allowed to create, delete, obtain details, and update a workgroup test_workgroup.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "athena:CreateNamedQuery",
                "athena:GetNamedQuery",
                "athena:BatchGetNamedQuery",
                "athena:ListNamedQueries",
                "athena:DeleteNamedQuery",
                "athena:CreatePreparedStatement",
                "athena:GetPreparedStatement",
                "athena:ListPreparedStatements",
                "athena:UpdatePreparedStatement",
                "athena:DeletePreparedStatement"
            ],
            "Resource": [
                "arn:aws:athena:us-east-1:123456789012:workgroup/workgroupA"
            ]
        }
    ]
}
```
Example Example policy for listing workgroups

The following policy allows all users to list all workgroups:

```json
{
   "Version": "2012-10-17",
   "Statement": [ 
      {
         "Effect": "Allow",
         "Action": [ "athena:ListWorkGroups" ],
         "Resource": "*"
      }
   ]
}
```

Example Example policy for running and stopping queries in a specific workgroup

In this policy, a user is allowed to run queries in the workgroup:

```json
{
   "Version": "2012-10-17",
   "Statement": [ 
      {
         "Effect": "Allow",
         "Action": [ "athena:StartQueryExecution", "athena:StopQueryExecution" ],
      }
   ]
}
```

Example Example policy for working with named queries in a specific workgroup

In the following policy, a user has permissions to create, delete, and obtain information about named queries in the specified workgroup:

```json
{
   "Version": "2012-10-17",
   "Statement": [ 
      {
         "Effect": "Allow",
         "Action": [ "athena:CreateNamedQuery", "athena:GetNamedQuery", "athena:DeleteNamedQuery" ],
      }
   ]
}
```
Workgroup settings

Each workgroup has the following settings:

- A unique name. It can contain from 1 to 128 characters, including alphanumeric characters, dashes, and underscores. After you create a workgroup, you cannot change its name. You can, however, create a new workgroup with the same settings and a different name.

- Settings that apply to all queries running in the workgroup. They include:
  - A location in Amazon S3 for storing query results for all queries that run in this workgroup. This location must exist before you specify it for the workgroup when you create it. For information about creating an Amazon S3 bucket, see Create a bucket.
  - Bucket owner control of query results – Whether the owner of the Amazon S3 query results bucket has full control over new objects that are written to the bucket. For example, if your query result location is owned by another account, you can grant ownership and full control over your query results to the other account.
  - Expected bucket owner – The ID of the AWS account that you expect to be the owner of the query results bucket. This is an added security measure. If the account ID of the bucket owner does not match the ID that you specify here, attempts to output to the bucket will fail. For in-depth information, see Verifying bucket ownership with bucket owner condition in the Amazon S3 User Guide.

  Note
  The expected bucket owner setting applies only to the Amazon S3 output location that you specify for Athena query results. It does not apply to other Amazon S3 locations like data source locations in external Amazon S3 buckets, CTAS and INSERT INTO destination table locations, UNLOAD statement output locations, operations to spill buckets for federated queries, or SELECT queries run against a table in another account.

  - An encryption setting, if you use encryption for all workgroup queries. You can encrypt only all queries in a workgroup, not just some of them. It is best to create separate workgroups to contain queries that are either encrypted or not encrypted.

In addition, you can override client-side settings (p. 638). Before the release of workgroups, you could specify results location and encryption options as parameters in the JDBC or ODBC driver, or in the Properties tab in the Athena console. These settings could also be specified directly via the API operations. These settings are known as "client-side settings". With workgroups, you can configure these settings at the workgroup level and enforce control over them. This spares your users from setting them individually. If you select the Override Client-Side Settings, queries use the workgroup settings and ignore the client-side settings.

If Override Client-Side Settings is selected, the user is notified on the console that their settings have changed. If workgroup settings are enforced this way, users can omit corresponding client-side settings. In this case, if you run queries in the console, the workgroup’s settings are used for them even if any queries have client-side settings. Also, if you run queries in this workgroup through the command line interface, API operations, or the drivers, any settings that you specified are overwritten by the workgroup's settings. This affects the query results location and encryption. To check which settings are used for the workgroup, view workgroup's details (p. 643).

You can also set query limits (p. 647) for queries in workgroups.

Workgroup settings override client-side settings

The Create workgroup and Edit workgroup dialogs have a field titled Override client-side settings. This field is unselected by default. Depending on whether you select it, Athena does the following:

- If Override client-side settings is not selected, workgroup settings are not enforced. In this case, for all queries that run in this workgroup, Athena uses the client-side settings for query results location,
expected bucket owner, encryption, and control of objects written to the query results bucket. Each user can specify client-side settings in the Settings menu on the console. If the client-side settings are not used, the workgroup-wide settings apply, but are not enforced. Also, if you run queries in this workgroup through the API operations, the command line interface, or the JDBC and ODBC drivers, and specify your query results location and encryption there, your queries continue using those settings.

- **If Override client-side settings** is selected, Athena uses the workgroup-wide settings for query results location, expected bucket owner, encryption, and control of objects written to the query results bucket. It also overrides any other settings that you specified for the query in the console, by using the API operations, or with the drivers. This affects you only if you run queries in this workgroup. If you do, workgroup settings are used.

If you override client-side settings, then the next time that you or any workgroup user open the Athena console, Athena notifies you that queries in this workgroup use workgroup's settings, and prompts you to acknowledge this change.

**Important**
If you run queries through the API operations, the command line interface, or the JDBC and ODBC drivers, and have not updated your settings to match those of the workgroup, your queries run, but use the workgroup's settings. For consistency, we recommend that you omit client-side settings in this case or update your query settings to match the workgroup's settings for the results location, expected bucket owner, encryption, and control of objects written to the query results bucket. To check which settings are used for the workgroup, view workgroup's details (p. 643).

### Managing workgroups

In the [https://console.aws.amazon.com/athena/](https://console.aws.amazon.com/athena/), you can perform the following tasks:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Create a workgroup (p. 640)</strong></td>
<td>Create a new workgroup.</td>
</tr>
<tr>
<td><strong>Edit a workgroup (p. 642)</strong></td>
<td>Edit a workgroup and change its settings. You cannot change a workgroup's name, but you can create a new workgroup with the same settings and a different name.</td>
</tr>
<tr>
<td><strong>View the workgroup's details (p. 643)</strong></td>
<td>View the workgroup's details, such as its name, description, data usage limits, location of query results, expected query results bucket owner, encryption, and control of objects written to the query results bucket. You can also verify whether this workgroup enforces its settings, if <strong>Override client-side settings</strong> is checked.</td>
</tr>
<tr>
<td><strong>Delete a workgroup (p. 643)</strong></td>
<td>Delete a workgroup. If you delete a workgroup, query history, saved queries, the workgroup's settings and per-query data limit controls are deleted. The workgroup-wide data limit controls remain in CloudWatch, and you can delete them individually. The primary workgroup cannot be deleted.</td>
</tr>
<tr>
<td><strong>Switch workgroups (p. 643)</strong></td>
<td>Switch between workgroups to which you have access.</td>
</tr>
<tr>
<td><strong>Copy a saved query between workgroups (p. 644)</strong></td>
<td>Copy a saved query between workgroups. You might want to do this if, for example, you created a query in a preview workgroup and you want to make it available in a nonpreview workgroup.</td>
</tr>
</tbody>
</table>
Create a workgroup

Creating a workgroup requires permissions to CreateWorkgroup API actions. See Access to workgroups and tags (p. 553) and IAM policies for accessing workgroups (p. 632). If you are adding tags, you also need to add permissions to TagResource. See Tag policy examples for workgroups (p. 664).

To create a workgroup in the console

1. If the console navigation pane is not visible, choose the expansion menu on the left.

2. In the Athena console navigation pane, choose Workgroups.

3. On the Workgroups page, choose Create workgroup.

4. On the Create workgroup page, fill in the fields as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workgroup name</td>
<td>Required. Enter a unique name for your workgroup. Use 1 - 128 characters. (A-Z,a-z,0-9,-,_.). This name cannot be changed.</td>
</tr>
<tr>
<td>Description</td>
<td>Optional. Enter a description for your workgroup. It can contain up to 1024 characters.</td>
</tr>
<tr>
<td>Update query engine</td>
<td>Choose how you want to update your workgroup when a new Athena engine version is released. You can let Athena decide when to update your workgroup or manually choose an engine version. For more information, see Athena engine versioning (p. 669).</td>
</tr>
</tbody>
</table>
| Location of query result | Optional. Enter a path to an Amazon S3 bucket or prefix. This bucket and prefix must exist before you can specify them. **Note**  
If you run queries in the console, specifying the query results location is optional. If you don't specify it for the workgroup |
### Managing workgroups

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>or in Settings, Athena uses the default query result location. If you run queries with the API or the drivers, you must specify query results location in at least one of the two places: for individual queries with OutputLocation, or for the workgroup, with WorkGroupConfiguration.</td>
<td></td>
</tr>
<tr>
<td><strong>Expected bucket owner</strong></td>
<td>Optional. Enter the ID of the AWS account that you expect to be the owner of the output location bucket. This is an added security measure. If the account ID of the bucket owner does not match the ID that you specify here, attempts to output to the bucket will fail. For in-depth information, see Verifying bucket ownership with bucket owner condition in the Amazon S3 User Guide.</td>
</tr>
<tr>
<td><strong>Note</strong></td>
<td>The expected bucket owner setting applies only to the Amazon S3 output location that you specify for Athena query results. It does not apply to other Amazon S3 locations like data source locations in external Amazon S3 buckets, CTAS and INSERT INTO destination table locations, UNLOAD statement output locations, operations to spill buckets for federated queries, or SELECT queries run against a table in another account.</td>
</tr>
<tr>
<td><strong>Encrypt query results</strong></td>
<td>Optional. Encrypt results stored in Amazon S3. If selected, all queries in the workgroup are encrypted.</td>
</tr>
<tr>
<td></td>
<td>If selected, you can select the Encryption type, the Encryption key and enter the KMS Key ARN.</td>
</tr>
<tr>
<td></td>
<td>If you don't have the key, open the AWS KMS console to create it. For more information, see Creating keys in the AWS Key Management Service Developer Guide.</td>
</tr>
<tr>
<td><strong>Assign bucket owner full control over query results</strong></td>
<td>This field is unselected by default. If you select it and ACLs are enabled for the query result location bucket, you grant full control access over query results to the bucket owner. For example, if your query result location is owned by another account, you can use this option to grant ownership and full control over your query results to the other account.</td>
</tr>
<tr>
<td></td>
<td>If the bucket's S3 Object Ownership setting is Bucket owner preferred, the bucket owner also owns all query result objects written from this workgroup. For example, if an external account's workgroup enables this option and sets its query result location to your account's Amazon S3 bucket which has an S3 Object Ownership setting of Bucket owner preferred, you own and have full control access over the external workgroup's query results.</td>
</tr>
<tr>
<td></td>
<td>Selecting this option when the query result bucket's S3 Object Ownership setting is Bucket owner enforced has no effect. For more information, see Object ownership settings in the Amazon S3 User Guide.</td>
</tr>
<tr>
<td><strong>Metrics</strong></td>
<td>This field is selected by default. Publish query metrics to CloudWatch. See Monitoring Athena queries with CloudWatch metrics (p. 648).</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Override client-side settings</td>
<td>This field is unselected by default. If you select it, workgroup settings apply to all queries in the workgroup and override client-side settings. For more information, see Workgroup settings override client-side settings (p. 638).</td>
</tr>
<tr>
<td>Requester Pays S3 buckets</td>
<td>Optional. Choose Enable queries on Requester Pays buckets in Amazon S3 if workgroup users will run queries on data stored in Amazon S3 buckets that are configured as Requester Pays. The account of the user running the query is charged for applicable data access and data transfer fees associated with the query. For more information, see Requester Pays buckets in the Amazon Simple Storage Service User Guide.</td>
</tr>
<tr>
<td>Per query data usage control</td>
<td>Optional. Sets the limit for the maximum amount of data a query is allowed to scan. You can set only one per query limit for a workgroup. The limit applies to all queries in the workgroup and if query exceeds the limit, it will be cancelled. For more information, see Setting data usage control limits (p. 655).</td>
</tr>
<tr>
<td>Workgroup data usage alerts</td>
<td>Optional. Set multiple alert thresholds when queries running in this workgroup scan a specified amount of data within a specific period. Alerts are implemented using Amazon CloudWatch alarms and applies to all queries in the workgroup. For more information, see Using Amazon CloudWatch alarms in the Amazon CloudWatch User Guide.</td>
</tr>
<tr>
<td>Tags</td>
<td>Optional. Add one or more tags to a workgroup. A tag is a label that you assign to an Athena workgroup resource. It consists of a key and a value. Use best practices for AWS tagging strategies to create a consistent set of tags and categorize workgroups by purpose, owner, or environment. You can also use tags in IAM policies, and to control billing costs. Do not use duplicate tag keys the same workgroup. For more information, see Tagging resources (p. 658).</td>
</tr>
</tbody>
</table>

5. Choose Create workgroup. The workgroup appears in the list on the Workgroups page.

You can also use the CreateWorkGroup API operation to create a workgroup.

**Important**
After you create workgroups, create IAM policies for accessing workgroups (p. 632) IAM that allow you to run workgroup-related actions.

## Edit a workgroup

Editing a workgroup requires permissions to UpdateWorkgroup API operations. See Access to workgroups and tags (p. 553) and IAM policies for accessing workgroups (p. 632). If you are adding or editing tags, you also need to have permissions to TagResource. See Tag policy examples for workgroups (p. 664).

### To edit a workgroup in the console

1. In the Athena console navigation pane, choose Workgroups.
2. On the Workgroups page, select the button for the workgroup that you want to edit.
3. Choose Actions, Edit.
4. Change the fields as needed. For the list of fields, see Create workgroup (p. 640). You can change all fields except for the workgroup’s name. If you need to change the name, create another workgroup with the new name and the same settings.
5. Choose **Save changes**. The updated workgroup appears in the list on the **Workgroups** page.

**View the workgroup's details**

For each workgroup, you can view its details. The details include the workgroup's name, description, whether it is enabled or disabled, and the settings used for queries that run in the workgroup, which include the location of the query results, expected bucket owner, encryption, and control of objects written to the query results bucket. If a workgroup has data usage limits, they are also displayed.

**To view the workgroup's details**

1. In the Athena console navigation pane, choose **Workgroups**.
2. On the **Workgroups** page, choose the link of the workgroup that you want to view. The **Overview Details** page for the workgroup displays.

**Delete a workgroup**

You can delete a workgroup if you have permissions to do so. The primary workgroup cannot be deleted.

If you have permissions, you can delete an empty workgroup at any time. You can also delete a workgroup that contains saved queries. In this case, before proceeding to delete a workgroup, Athena warns you that saved queries are deleted.

If you delete a workgroup while you are in it, the console switches focus to the primary workgroup. If you have access to it, you can run queries and view its settings.

If you delete a workgroup, its settings and per-query data limit controls are deleted. The workgroup-wide data limit controls remain in CloudWatch, and you can delete them there if needed.

**Important**

Before deleting a workgroup, ensure that its users also belong to other workgroups where they can continue to run queries. If the users' IAM policies allowed them to run queries only in this workgroup, and you delete it, they no longer have permissions to run queries. For more information, see [Example Policy for Running Queries in the Primary Workgroup](#) (p. 636).

**To delete a workgroup in the console**

1. In the Athena console navigation pane, choose **Workgroups**.
2. On the **Workgroups** page, select the button for the workgroup that you want to delete.
3. Choose **Actions, Delete**.
4. At the **Delete workgroup** confirmation prompt, enter the name of the workgroup, and then choose **Delete**.

To delete a workgroup with the API operation, use the **DeleteWorkGroup** action.

**Switch workgroups**

You can switch from one workgroup to another if you have permissions to both of them.

You can open up to ten query tabs within each workgroup. When you switch between workgroups, your query tabs remain open for up to three workgroups.

**To switch workgroups**

1. In the Athena console, use the **Workgroup** option on the upper right to choose a workgroup.
2. If the **Workgroup workgroup-name settings** dialog box appears, choose **Acknowledge**.

The **Workgroup** option shows the name of the workgroup that you switched to. You can now run queries in this workgroup.

### Copy a saved query between workgroups

Currently, the Athena console does not have an option to copy a saved query from one workgroup to another directly, but you can perform the same task manually by using the following procedure.

**To copy a saved query between workgroups**

1. In the Athena console, from the workgroup that you want to copy the query from, choose the **Saved queries** tab.
2. Choose the link of the saved query that you want to copy. Athena opens the query in the query editor.
3. In the query editor, select the query text, and then press Ctrl+C to copy it.
4. **Switch** (p. 643) to the destination workgroup, or **create a workgroup** (p. 640), and then switch to it.
5. Open a new tab in the query editor, and then press Ctrl+V to paste the text into the new tab.
6. In the query editor, choose **Save as** to save the query in the destination workgroup.
7. In the **Choose a name** dialog box, enter a name for the query and an optional description.
8. Choose **Save**.

### Enable and disable a workgroup

If you have permissions to do so, you can enable or disable workgroups in the console, by using the API operations, or with the JDBC and ODBC drivers.

**To enable or disable a workgroup**

1. In the Athena console navigation pane, choose **Workgroups**.
2. On the **Workgroups** page, choose the link for the workgroup.
3. On the upper right, choose **Enable workgroup** or **Disable workgroup**.
4. At the confirmation prompt, choose **Enable** or **Disable**. If you disable a workgroup, its users cannot run queries in it, or create new named queries. If you enable a workgroup, users can use it to run queries.

### Specify a workgroup in which to run queries

To specify a workgroup to use, you must have permissions to the workgroup.

**To specify the workgroup to use**

1. Make sure your permissions allow you to run queries in a workgroup that you intend to use. For more information, see the section called "IAM policies for accessing workgroups" (p. 632).
2. To specify the workgroup, use one of these options:
   - If you are using the Athena console, set the workgroup by **switching workgroups** (p. 643).
   - If you are using the Athena API operations, specify the workgroup name in the API action. For example, you can set the workgroup name in **StartQueryExecution**, as follows:
StartQueryExecutionRequest startQueryExecutionRequest = new StartQueryExecutionRequest()
  .withQueryString(ExampleConstants.ATHENA_SAMPLE_QUERY)
  .withQueryExecutionContext(queryExecutionContext)
  .withWorkGroup(WorkgroupName)

• If you are using the JDBC or ODBC driver, set the workgroup name in the connection string using
  the Workgroup configuration parameter. The driver passes the workgroup name to Athena.
  Specify the workgroup parameter in the connection string as in the following example:

```
jdbc:awsathena://AwsRegion=<AWSREGION>;UID=<ACCESSKEY>;
PWD=<SECRETKEY>;S3OutputLocation=s3://<athena-output>-<AWSREGION>/;
Workgroup=<WORKGROUPNAME>;
```

For more information, search for "Workgroup" in the driver documentation link included in JDBC
driver documentation (p. 185).

Athena workgroup APIs

The following are some of the REST API operations used for Athena workgroups. In all of the following
operations except for ListWorkGroups, you must specify a workgroup. In other operations, such as
StartQueryExecution, the workgroup parameter is optional and the operations are not listed here.
For the full list of operations, see Amazon Athena API Reference.

• CreateWorkGroup
• DeleteWorkGroup
• GetWorkGroup
• ListWorkGroups
• UpdateWorkGroup

Troubleshooting workgroups

Use the following tips to troubleshoot workgroups.

• Check permissions for individual users in your account. They must have access to the location for query
  results, and to the workgroup in which they want to run queries. If they want to switch workgroups,
  they too need permissions to both workgroups. For information, see IAM policies for accessing
  workgroups (p. 632).

• Pay attention to the context in the Athena console, to see in which workgroup you are going to run
  queries. If you use the driver, make sure to set the workgroup to the one you need. For information,
  see the section called “Specify a workgroup in which to run queries” (p. 644).

• If you use the API or the drivers to run queries, you must specify the query results location using one
  of the following ways: for individual queries, use OutputLocation (client-side). In the workgroup, use
  WorkGroupConfiguration. If the location is not specified in either way, Athena issues an error at query
  runtime.

• If you override client-side settings with workgroup settings, you may encounter errors with query
  result location. For example, a workgroup’s user may not have permissions to the workgroup’s location
  in Amazon S3 for storing query results. In this case, add the necessary permissions.

• Workgroups introduce changes in the behavior of the API operations. Calls to the following existing
  API operations require that users in your account have resource-based permissions in IAM to the
workgroups in which they make them. If no permissions to the workgroup and to workgroup actions exist, the following API actions throw AccessDeniedException: CreateNamedQuery, DeleteNamedQuery, GetNamedQuery, ListNamedQueries, StartQueryExecution, StopQueryExecution, ListQueryExecutions, GetQueryExecution, GetQueryResults, and GetQueryResultsStream (this API action is only available for use with the driver and is not exposed otherwise for public use). For more information, see Actions, resources, and condition keys for Amazon Athena in the Service Authorization Reference.

Calls to the BatchGetQueryExecution and BatchGetNamedQuery API operations return information only about queries that run in workgroups to which users have access. If the user has no access to the workgroup, these API operations return the unauthorized query IDs as part of the unprocessed IDs list. For more information, see the section called "Athena workgroup APIs" (p. 645).

• If the workgroup in which a query will run is configured with an enforced query results location (p. 638), do not specify an external_location for the CTAS query. Athena issues an error and fails a query that specifies an external_location in this case. For example, this query fails, if you override client-side settings for query results location, enforcing the workgroup to use its own location: CREATE TABLE <DB>.<TABLE1> WITH (format='Parquet', external_location='s3://my_test/test/') AS SELECT * FROM <DB>.<TABLE2> LIMIT 10;

You may see the following errors. This table provides a list of some of the errors related to workgroups and suggests solutions.

**Workgroup errors**

<table>
<thead>
<tr>
<th>Error</th>
<th>Occurs when...</th>
</tr>
</thead>
<tbody>
<tr>
<td>query state CANCELED. Bytes scanned limit was exceeded.</td>
<td>A query hits a per-query data limit and is canceled. Consider rewriting the query so that it reads less data, or contact your account administrator.</td>
</tr>
<tr>
<td>User: arn:aws:iam::123456789012:user/abc is not authorized to perform: athena:StartQueryExecution on resource: arn:aws:athena:us-east-1:123456789012:workgroup/workgroupname</td>
<td>A user runs a query in a workgroup, but does not have access to it. Update your policy to have access to the workgroup.</td>
</tr>
<tr>
<td>INVALID_INPUT. WorkGroup &lt;name&gt; is disabled.</td>
<td>A user runs a query in a workgroup, but the workgroup is disabled. Your workgroup could be disabled by your administrator. It is possible also that you don't have access to it. In both cases, contact an administrator who has access to modify workgroups.</td>
</tr>
<tr>
<td>INVALID_INPUT. WorkGroup &lt;name&gt; is not found.</td>
<td>A user runs a query in a workgroup, but the workgroup does not exist. This could happen if the workgroup was deleted. Switch to another workgroup to run your query.</td>
</tr>
<tr>
<td>InvalidRequestException: when calling the StartQueryExecution operation: No output location provided. An output location is required either through the Workgroup result configuration setting or as an API input.</td>
<td>A user runs a query with the API without specifying the location for query results. You must set the output location for query results using one of the two ways: either for individual queries, using OutputLocation (client-side), or in the workgroup, using WorkGroupConfiguration.</td>
</tr>
</tbody>
</table>
Controlling costs and monitoring queries with CloudWatch metrics and events

Workgroups allow you to set data usage control limits per query or per workgroup, set up alarms when those limits are exceeded, and publish query metrics to CloudWatch.

In each workgroup, you can:

- Configure Data usage controls per query and per workgroup, and establish actions that will be taken if queries breach the thresholds.
- View and analyze query metrics, and publish them to CloudWatch. If you create a workgroup in the console, the setting for publishing the metrics to CloudWatch is selected for you. If you use the API operations, you must enable publishing the metrics (p. 647). When metrics are published, they are displayed under the Metrics tab in the Workgroups panel. Metrics are disabled by default for the primary workgroup.

Video

The following video shows how to create custom dashboards and set alarms and triggers on metrics in CloudWatch. You can use pre-populated dashboards directly from the Athena console to consume these query metrics.

Monitoring Amazon Athena queries using Amazon CloudWatch

Topics

- Enabling CloudWatch query metrics (p. 647)
- Monitoring Athena queries with CloudWatch metrics (p. 648)
- Monitoring Athena queries with CloudWatch events (p. 651)
- Monitoring Athena usage metrics (p. 654)
- Setting data usage control limits (p. 655)

Enabling CloudWatch query metrics

When you create a workgroup in the console, the setting for publishing query metrics to CloudWatch is selected by default.
To enable or disable query metrics in the Athena console for a workgroup

2. If the console navigation pane is not visible, choose the expansion menu on the left.
3. In the navigation pane, choose Workgroups.
4. Choose the link of the workgroup that you want to modify.
5. On the details page for the workgroup, choose Edit.
6. In the Settings section, select or clear Publish query metrics to AWS CloudWatch.

If you use API operations, the command line interface, or the client application with the JDBC driver to create workgroups, to enable publishing of query metrics, set PublishCloudWatchMetricsEnabled to true in WorkGroupConfiguration. The following example shows only the metrics configuration and omits other configuration:

"WorkGroupConfiguration": {
    "PublishCloudWatchMetricsEnabled": "true"
    ....
}

Monitoring Athena queries with CloudWatch metrics

Athena publishes query-related metrics to Amazon CloudWatch, when the publish query metrics to CloudWatch (p. 647) option is selected. You can create custom dashboards, set alarms and triggers on metrics in CloudWatch, or use pre-populated dashboards directly from the Athena console.

When you enable query metrics for queries in workgroups, the metrics are displayed within the Metrics tab in the Workgroups panel, for each workgroup in the Athena console.

Athena publishes the following metrics to the CloudWatch console:

• EngineExecutionTime – in milliseconds
• ProcessedBytes – the total amount of data scanned per DML query
• QueryPlanningTime – in milliseconds
• QueryQueueTime – in milliseconds
• ServiceProcessingTime – in milliseconds
• TotalExecutionTime – in milliseconds, for DDL and DML queries

These metrics have the following dimensions:

• QueryState – QUEUED, RUNNING, SUCCEEDED, FAILED, or CANCELED
Monitoring Athena queries with CloudWatch metrics

- QueryType – DML or DDL
- WorkGroup – name of the workgroup

For more information, see the List of CloudWatch metrics and dimensions for Athena (p. 650) later in this topic. For information about Athena usage metrics, see Monitoring Athena usage metrics (p. 654).

To view query metrics for a workgroup in the console

2. If the console navigation pane is not visible, choose the expansion menu on the left.

3. In the navigation pane, choose Workgroups.
4. Choose the workgroup that you want from the list, and then choose the Metrics tab.

The metrics dashboard displays.

Note
If you just recently enabled metrics for the workgroup and/or there has been no recent query activity, the graphs on the dashboard may be empty. Query activity is retrieved from CloudWatch depending on the interval that you specify in the next step.

5. In the Metrics section, choose the metrics interval that Athena should use to fetch the query metrics from CloudWatch, or specify a custom interval.

6. To refresh the displayed metrics, choose the refresh icon.
7. Click the arrow next to the refresh icon to choose how frequently you want the metrics display to be updated.

To view metrics in the Amazon CloudWatch console
2. In the navigation pane, choose Metrics, All metrics.
3. Select the AWS/Athena namespace.

To view metrics with the CLI
- To list the metrics for Athena, open a command prompt, and use the following command:

```
aws cloudwatch list-metrics --namespace "AWS/Athena"
```
- To list all available metrics, use the following command:

```
aws cloudwatch list-metrics
```

List of CloudWatch metrics and dimensions for Athena
If you've enabled CloudWatch metrics in Athena, it sends the following metrics to CloudWatch per workgroup. The metrics use the AWS/Athena namespace.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EngineExecutionTime</td>
<td>The number of milliseconds that the query took to run.</td>
</tr>
<tr>
<td>ProcessedBytes</td>
<td>The number of bytes that Athena scanned per DML query. For queries that were canceled (either by the users, or automatically, if they reached the limit), this includes the amount of data scanned before the cancellation time. This metric is not reported for DDL queries.</td>
</tr>
<tr>
<td>QueryPlanningTime</td>
<td>The number of milliseconds that Athena took to plan the query processing flow. This includes the time spent retrieving table partitions from the data source. Note that because the query engine performs the query planning, query planning time is a subset of EngineExecutionTime.</td>
</tr>
<tr>
<td>Metric name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>QueryQueueTime</td>
<td>The number of milliseconds that the query was in the query queue waiting for resources. Note that if transient errors occur, the query can be automatically added back to the queue.</td>
</tr>
<tr>
<td>ServiceProcessingTime</td>
<td>Number of milliseconds that Athena took to process the query results after the query engine finished running the query.</td>
</tr>
<tr>
<td>TotalExecutionTime</td>
<td>The number of milliseconds that Athena took to run a DDL or DML query. TotalExecutionTime includes QueryQueueTime, QueryPlanningTime, EngineExecutionTime, and ServiceProcessingTime.</td>
</tr>
</tbody>
</table>

CloudWatch metrics for Athena have the following dimensions.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QueryState</td>
<td>The query state.</td>
</tr>
<tr>
<td></td>
<td>Valid statistics: QUEUED, RUNNING, SUCCEEDED, FAILED, or CANCELED.</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong> Athena automatically retries your queries in cases of certain transient errors. As a result, you may see the query state transition from RUNNING or FAILED to QUEUED.</td>
</tr>
<tr>
<td>QueryType</td>
<td>The query type.</td>
</tr>
<tr>
<td></td>
<td>Valid statistics: DDL or DML.</td>
</tr>
<tr>
<td>WorkGroup</td>
<td>The name of the workgroup.</td>
</tr>
</tbody>
</table>

**Monitoring Athena queries with CloudWatch events**

You can use Amazon Athena with Amazon CloudWatch to receive real-time notifications regarding the state of your queries. When a query you have submitted transitions states, Athena publishes an event to CloudWatch Events containing information about that query state transition. You can write simple rules for events that are of interest to you and take automated actions when an event matches a rule. For example, you can create a rule that invokes an AWS Lambda function when a query reaches a terminal state. Events are emitted on a best effort basis.

Before you create event rules for Athena, you should do the following:

- Familiarize yourself with events, rules, and targets in CloudWatch Events. For more information, see [What is Amazon CloudWatch events?](#) For more information about how to set up rules, see [Getting started with CloudWatch Events](#).
- Create the target or targets to use in your event rules.

**Note**

Athena currently offers one type of event, Athena Query State Change, but may add other event types and details. If you are programmatically deserializing event JSON data, make sure that your application is prepared to handle unknown properties if additional properties are added.
Athena event format

The following is the basic pattern for an Amazon Athena event.

```
{
  "source": [
    "aws.athena"
  ],
  "detail-type": [
    "Athena Query State Change"
  ],
  "detail": {
    "currentState": [
      "SUCCEEDED"
    ]
  }
}
```

Athena query state change event

The following example shows an Athena Query State Change event with the `currentState` value of `SUCCEEDED`.

```
{
  "version": "0",
  "id": "abcdef00-1234-5678-9abc-def012345678",
  "detail-type": "Athena Query State Change",
  "source": "aws.athena",
  "account": "123456789012",
  "time": "2019-10-06T09:30:10Z",
  "region": "us-east-1",
  "resources": [
  ],
  "detail": {
    "versionId": "0",
    "currentState": "SUCCEEDED",
    "previousState": "RUNNING",
    "statementType": "DDL",
    "queryExecutionId": "01234567-0123-0123-0123-012345678901",
    "workgroupName": "primary",
    "sequenceNumber": "3"
  }
}
```

The following example shows an Athena Query State Change event with the `currentState` value of `FAILED`. The `athenaError` block appears only when `currentState` is `FAILED`. For information about the values for `errorCategory` and `errorType`, see Athena error catalog (p. 777).

```
{
  "version": "0",
  "id": "abcdef00-1234-5678-9abc-def012345678",
  "detail-type": "Athena Query State Change",
  "source": "aws.athena",
  "account": "123456789012",
  "time": "2019-10-06T09:30:10Z",
  "region": "us-east-1",
  "resources": [
  ],
  "detail": {
```

652
"athenaError": {  
    "errorCategory": 2.0, //Value depends on nature of exception  
    "errorType": 1306.0, //Type depends on nature of exception  
    "errorMessage": "Amazon S3 bucket not found", //Message depends on nature of exception  
    "retryable":false //Retryable value depends on nature of exception  
},  
"versionId":"0",  
"currentState": "FAILED",  
"previousState": "RUNNING",  
"statementType": "DML",  
"queryExecutionId":"01234567-0123-0123-0123-012345678901",  
"workgroupName": "primary",  
"sequenceNumber": "3"  
}  

Output properties

The JSON output includes the following properties.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>athenaError</td>
<td>Appears only when currentState is FAILED. Contains information about the error that occurred, including the error category, error type, error message, and whether the action that led to the error can be retried. Values for each of these fields depend on the nature of the error. For information about the values for errorCategory and errorType, see Athena error catalog (p. 777).</td>
</tr>
<tr>
<td>versionId</td>
<td>The version number for the detail object's schema.</td>
</tr>
<tr>
<td>currentState</td>
<td>The state that the query transitioned to at the time of the event.</td>
</tr>
<tr>
<td>previousState</td>
<td>The state that the query transitioned from at the time of the event.</td>
</tr>
<tr>
<td>statementType</td>
<td>The type of query statement that was run.</td>
</tr>
<tr>
<td>queryExecutionId</td>
<td>The unique identifier for the query that ran.</td>
</tr>
<tr>
<td>workgroupName</td>
<td>The name of the workgroup in which the query ran.</td>
</tr>
<tr>
<td>sequenceNumber</td>
<td>A monotonically increasing number that allows for deduplication and ordering of incoming events that involve a single query that ran. When duplicate events are published for the same state transition, the sequenceNumber value is the same. When a query experiences a state transition more than once, such as queries that experience rare requeuing, you can use sequenceNumber to order events with identical currentState and previousState values.</td>
</tr>
</tbody>
</table>

Example

The following example publishes events to an Amazon SNS topic to which you have subscribed. When Athena is queried, you receive an email. The example assumes that the Amazon SNS topic exists and that you have subscribed to it.
To publish Athena events to an Amazon SNS topic

1. Create the target for your Amazon SNS topic. Give the CloudWatch Events Service Principal events.amazonaws.com permission to publish to your Amazon SNS topic, as in the following example.

   ```json
   {
     "Effect":"Allow",
     "Principal":{
       "Service":"events.amazonaws.com"
     },
     "Action":"sns:Publish",
     "Resource":"arn:aws:sns:us-east-1:1111111111:your-sns-topic"
   }
   ```

2. Use the AWS CLI `events put-rule` command to create a rule for Athena events, as in the following example.

   ```bash
   aws events put-rule --name {ruleName} --event-pattern '{"source": ["aws.athena"]}'
   ```

3. Use the AWS CLI `events put-targets` command to attach the Amazon SNS topic target to the rule, as in the following example.

   ```bash
   aws events put-targets --rule {ruleName} --targets Id=1,Arn=arn:aws:sns:us-east-1:1111111111:your-sns-topic
   ```

4. Query Athena and observe the target being invoked. You should receive corresponding emails from the Amazon SNS topic.

Monitoring Athena usage metrics

You can use CloudWatch usage metrics to provide visibility into your how your account uses resources by displaying your current service usage on CloudWatch graphs and dashboards.

For Athena, usage availability metrics correspond to AWS service quotas for Athena. You can configure alarms that alert you when your usage approaches a service quota. For more information about Athena service quotas, see Service Quotas (p. 785). For more information about AWS usage metrics, see AWS usage metrics in the Amazon CloudWatch User Guide.

Athena publishes the following metrics in the AWS/Usage namespace.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActiveQueryCount</td>
<td>The sum of all queued and executing queries for an account, separated by query type (DML or DDL). Sum is the only useful statistic for this metric.</td>
</tr>
</tbody>
</table>

   This metric publishes periodically every minute. If you are not running any queries, the metric reports nothing (not even 0). The metric publishes only if active queries are running at the time the metric is taken.

The following dimensions are used to refine the usage metrics that are published by Athena.
### Setting data usage control limits

Athena allows you to set two types of cost controls: per-query limit and per-workgroup limit. For each workgroup, you can set only one per-query limit and multiple per-workgroup limits.

- **The per-query control limit** specifies the total amount of data scanned per query. If any query that runs in the workgroup exceeds the limit, it is canceled. You can create only one per-query control limit in a workgroup and it applies to each query that runs in it. Edit the limit if you need to change it. For detailed steps, see [To create a per-query data usage control](p. 656).

- **The workgroup-wide data usage control limit** specifies the total amount of data scanned for all queries that run in this workgroup during the specified time period. You can create multiple limits per workgroup. The workgroup-wide query limit allows you to set multiple thresholds on hourly or daily aggregates on data scanned by queries running in the workgroup.

  If the aggregate amount of data scanned exceeds the threshold, you can push a notification to an Amazon SNS topic. To do this, you configure an Amazon SNS alarm and an action in the Athena console.

---

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>The name of the AWS service containing the resource. For Athena, the value for this dimension is Athena.</td>
</tr>
<tr>
<td>Type</td>
<td>The type of entity that's being reported. Currently, the only valid value for Athena usage metrics is Resource.</td>
</tr>
<tr>
<td>Resource</td>
<td>The type of resource that is running. The resource value for Athena query usage is ActiveQueryCount.</td>
</tr>
<tr>
<td>Class</td>
<td>The class of resource being tracked. For Athena, Class can be DML or DDL.</td>
</tr>
</tbody>
</table>

---

**Viewing Athena resource usage metrics in the CloudWatch console**

You can use the CloudWatch console to see a graph of Athena usage metrics and configure alarms that alert you when your usage approaches a service quota.

**To view Athena resource usage metrics**

2. In the navigation pane, choose Metrics, All metrics.
3. If you are in the AWS/Athena, namespace, choose All.
4. Choose Usage, and then choose By AWS Resource.

   The list of service quota usage metrics appears.
5. Select the check box that is next to Athena and ActiveQueryCount.

   The graph above displays your current usage of the AWS resource.

For information about adding service quotas to the graph and setting an alarm that notifies you if you approach the service quota, see [Visualizing your service quotas and setting alarms](p. 655) in the Amazon CloudWatch User Guide. For information about setting usage limits per workgroup, see [Setting data usage control limits](p. 655).
console to notify an administrator when the limit is breached. For detailed steps, see **To create a per-workgroup data usage control** (p. 657). You can also create an alarm and an action on any metric that Athena publishes from the CloudWatch console. For example, you can set an alert on a number of failed queries. This alert can trigger an email to an administrator if the number crosses a certain threshold. If the limit is exceeded, an action sends an Amazon SNS alarm notification to the specified users.

Other actions you can take:

- Invoke a Lambda function. For more information, see **Invoking Lambda functions using Amazon SNS notifications** in the *Amazon Simple Notification Service Developer Guide*.
- Disable the workgroup to stop any further queries from running. For steps, see **Enable and disable a workgroup** (p. 644).

The per-query and per-workgroup limits are independent of each other. A specified action is taken whenever either limit is exceeded. If two or more users run queries at the same time in the same workgroup, it is possible that each query does not exceed any of the specified limits, but the total sum of data scanned exceeds the data usage limit per workgroup. In this case, an Amazon SNS alarm is sent to the user.

**To create a per-query data usage control**

The per-query control limit specifies the total amount of data scanned per query. If any query that runs in the workgroup exceeds the limit, it is canceled. Canceled queries are charged according to **Amazon Athena pricing**.

**Note**

In the case of canceled or failed queries, Athena may have already written partial results to Amazon S3. In such cases, Athena does not delete partial results from the Amazon S3 prefix where results are stored. You must remove the Amazon S3 prefix with partial results. Athena uses Amazon S3 multipart uploads to write data Amazon S3. We recommend that you set the bucket lifecycle policy to end multipart uploads in cases when queries fail. For more information, see **Aborting incomplete multipart uploads using a bucket lifecycle policy** in the *Amazon Simple Storage Service User Guide*.

You can create only one per-query control limit in a workgroup and it applies to each query that runs in it. Edit the limit if you need to change it.

2. If the console navigation pane is not visible, choose the expansion menu on the left.
3. In the navigation pane, choose **Workgroups**.
4. Choose the name of the workgroup from the list.
5. On the **Data usage controls** tab, in the **Per query data usage control** section, choose **Manage**.
6. On the **Manage per query data usage control** page, specify the following values:
• For **Data limit**, specify a value between 10 MB (minimum) and 7 EB (maximum).

  **Note**
  These are limits imposed by the console for data usage controls within workgroups. They do not represent any query limits in Athena.

• For units, select the unit value from the drop-down list (for example, **Kilobytes KB** or **Exabytes EB**).

The default action is to cancel the query if it exceeds the limit. This setting cannot be changed.

7. Choose **Save**.

### To create or edit a per-workgroup data usage alert

You can set multiple alert thresholds when queries running in a workgroup scan a specified amount of data within a specific period. Alerts are implemented using Amazon CloudWatch alarms and apply to all queries in the workgroup. When a threshold is reached, you can have Amazon SNS send an email to users that you specify. Queries are not automatically canceled when a threshold is reached.

2. If the console navigation pane is not visible, choose the expansion menu on the left.
3. In the navigation pane, choose **Workgroups**.
4. Choose the name of the workgroup from the list.
5. Choose **Edit** to edit the workgroup’s settings.
6. Scroll down to and expand **Workgroup data usage alerts - optional**.
7. Choose **Add alert**.
8. For **Data usage threshold configuration**, specify values as follows:

   • For **Data threshold**, specify a number, and then select a unit value from the drop-down list.
   • For **Time period**, choose a time period from the drop-down list.
   • For **SNS topic selection**, choose an Amazon SNS topic from the drop-down list. Or, choose **Create SNS topic** to go directly to the Amazon SNS console, create the Amazon SNS topic, and set up a subscription for it for one of the users in your Athena account. For more information, see **Getting started with Amazon SNS** in the Amazon Simple Notification Service Developer Guide.
9. Choose **Add alert** if you are creating a new alert, or **Save** to save an existing alert.
Tagging Athena resources

A tag consists of a key and a value, both of which you define. When you tag an Athena resource, you assign custom metadata to it. You can use tags to categorize your AWS resources in different ways; for example, by purpose, owner, or environment. In Athena, workgroups and data catalogs are taggable resources. For example, you can create a set of tags for workgroups in your account that helps you track workgroup owners, or identify workgroups by their purpose. If you also enable the tags as cost allocation tags in the Billing and Cost Management console, costs associated with running queries appear in your Cost and Usage Report with that cost allocation tag. We recommend that you use AWS tagging best practices to create a consistent set of tags to meet your organization requirements.

You can work with tags using the Athena console or the API operations.

Topics

- Tag basics (p. 658)
- Tag restrictions (p. 658)
- Working with tags on workgroups in the console (p. 659)
- Using tag operations (p. 660)
- Tag-based IAM access control policies (p. 664)

Tag basics

A tag is a label that you assign to an Athena resource. Each tag consists of a key and an optional value, both of which you define.

Tags enable you to categorize your AWS resources in different ways. For example, you can define a set of tags for your account's workgroups that helps you track each workgroup owner or purpose.

You can add tags when creating a new Athena workgroup or data catalog, or you can add, edit, or remove tags from them. You can edit a tag in the console. To use API operations to edit a tag, remove the old tag and add a new one. If you delete a resource, any tags for the resource are also deleted.

Athena does not automatically assign tags to your resources. You can edit tag keys and values, and you can remove tags from a resource at any time. You can set the value of a tag to an empty string, but you can't set the value of a tag to null. Do not add duplicate tag keys to the same resource. If you do, Athena issues an error message. If you use the TagResource action to tag a resource using an existing tag key, the new tag value overwrites the old value.

In IAM, you can control which users in your Amazon Web Services account have permission to create, edit, remove, or list tags. For more information, see Tag-based IAM access control policies (p. 664).

For a complete list of Amazon Athena tag actions, see the API action names in the Amazon Athena API Reference.

You can use tags for billing. For more information, see Using tags for billing in the AWS Billing and Cost Management User Guide.

For more information, see Tag restrictions (p. 658).

Tag restrictions

Tags have the following restrictions:
• In Athena, you can tag workgroups and data catalogs. You cannot tag queries.
• The maximum number of tags per resource is 50. To stay within the limit, review and delete unused tags.
• For each resource, each tag key must be unique, and each tag key can have only one value. Do not add duplicate tag keys at the same time to the same resource. If you do, Athena issues an error message. If you tag a resource using an existing tag key in a separate TagResource action, the new tag value overwrites the old value.
• Tag key length is 1-128 Unicode characters in UTF-8.
• Tag value length is 0-256 Unicode characters in UTF-8.

Tagging operations, such as adding, editing, removing, or listing tags, require that you specify an ARN for the workgroup resource.
• Athena allows you to use letters, numbers, spaces represented in UTF-8, and the following characters: + - . _ : / @.
• Tag keys and values are case-sensitive.
• The "aws:" prefix in tag keys is reserved for AWS use. You can't edit or delete tag keys with this prefix. Tags with this prefix do not count against your per-resource tags limit.
• The tags you assign are available only to your Amazon Web Services account.

Working with tags on workgroups in the console

Using the Athena console, you can see which tags are in use by each workgroup in your account. You can view tags by workgroup only. You can also use the Athena console to apply, edit, or remove tags from one workgroup at a time.

You can search workgroups using the tags you created.

Topics
• Displaying tags for individual workgroups (p. 659)
• Adding and deleting tags on an individual workgroup (p. 660)

Displaying tags for individual workgroups

To display tags for an individual workgroup in the Athena console
2. If the console navigation pane is not visible, choose the expansion menu on the left.
3. On the navigation menu, choose Workgroups, and then choose the workgroup that you want.
4. Do one of the following:
• Choose the **Tags** tab. If the list of tags is long, use the search box.
• Choose **Edit**, and then scroll down to the **Tags** section.

## Adding and deleting tags on an individual workgroup

You can manage tags for an individual workgroup directly from the **Workgroups** tab.

**Note**

If you want users to add tags when they create a workgroup in the console or pass in tags when they use the **CreateWorkGroup** action, make sure that you give the users IAM permissions to the **TagResource** and **CreateWorkGroup** actions.

### To add a tag when you create a new workgroup

2. On the navigation menu, choose **Workgroups**.
3. Choose **Create workgroup** and fill in the values as needed. For detailed steps, see [Create a workgroup (p. 640)](#).
4. In the **Tags** section, add one or more tags by specifying keys and values. Do not add duplicate tag keys at the same time to the same workgroup. If you do, Athena issues an error message. For more information, see [Tag restrictions (p. 658)](#).
5. When you are done, choose **Create workgroup**.

### To add or edit a tag to an existing workgroup

2. In the navigation pane, choose **Workgroups**.
3. Choose the workgroup that you want to modify.
4. Do one of the following:
   • Choose the **Tags** tab, and then choose **Manage tags**.
   • Choose **Edit**, and then scroll down to the **Tags** section.
5. Specify a key and value for each tag. For more information, see [Tag restrictions (p. 658)](#).
6. Choose **Save**.

### To delete a tag from an individual workgroup

2. In the navigation pane, choose **Workgroups**.
3. Choose the workgroup that you want to modify.
4. Do one of the following:
   • Choose the **Tags** tab, and then choose **Manage tags**.
   • Choose **Edit**, and then scroll down to the **Tags** section.
5. In the list of tags, choose **Remove** for the tag that you want to delete, and then choose **Save**.

## Using tag operations

Use the following tag operations to add, remove, or list tags on a resource.
### API Operations for Managing Tags

<table>
<thead>
<tr>
<th>API</th>
<th>CLI</th>
<th>Action description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TagResource</td>
<td>tag-resource</td>
<td>Add or overwrite one or more tags on the resource that has the specified ARN.</td>
</tr>
<tr>
<td>UntagResource</td>
<td>untag-resource</td>
<td>Delete one or more tags from the resource that has the specified ARN.</td>
</tr>
<tr>
<td>ListTagsForResource</td>
<td>list-tags-for-resource</td>
<td>List one or more tags for the resource that has the specified ARN.</td>
</tr>
</tbody>
</table>

### Adding Tags When Creating a Resource

To add tags when you create a workgroup or data catalog, use the tags parameter with the CreateWorkGroup or CreateDataCatalog API operations or with the AWS CLI create-work-group or create-data-catalog commands.

### Managing tags using API operations

The examples in this section show how to use tag API operations to manage tags on workgroups and data catalogs. The examples are in the Java programming language.

#### Example TagResource

The following example adds two tags to the workgroup `workgroupA`:

```java
List<Tag> tags = new ArrayList<>();
tag.add(new Tag().withKey("tagKey1").withValue("tagValue1"));
tag.add(new Tag().withKey("tagKey2").withValue("tagValue2"));
TagResourceRequest request = new TagResourceRequest()
    .withTags(tags);
client.tagResource(request);
```

The following example adds two tags to the data catalog `datacatalogA`:

```java
List<Tag> tags = new ArrayList<>();
tag.add(new Tag().withKey("tagKey1").withValue("tagValue1"));
tag.add(new Tag().withKey("tagKey2").withValue("tagValue2"));
TagResourceRequest request = new TagResourceRequest()
    .withTags(tags);
client.tagResource(request);
```

**Note**

Do not add duplicate tag keys to the same resource. If you do, Athena issues an error message.

If you tag a resource using an existing tag key in a separate TagResource action, the new tag value overwrites the old value.

#### Example UntagResource

The following example removes `tagKey2` from the workgroup `workgroupA`:

```java
TagResourceRequest request = new TagResourceRequest()
    .withTags(tags);
client.tagResource(request);
```
The following example removes tagKey2 from the data catalog datacatalogA:

```java
List<String> tagKeys = new ArrayList<>();
tagKeys.add("tagKey2");

UntagResourceRequest request = new UntagResourceRequest()
    .withTagKeys(tagKeys);
client.untagResource(request);
```

Example ListTagsForResource

The following example lists tags for the workgroup workgroupA:

```java
ListTagsForResourceRequest request = new ListTagsForResourceRequest()
ListTagsForResourceResult result = client.listTagsForResource(request);
List<Tag> resultTags = result.getTags();
```

The following example lists tags for the data catalog datacatalogA:

```java
ListTagsForResourceRequest request = new ListTagsForResourceRequest()
ListTagsForResourceResult result = client.listTagsForResource(request);
List<Tag> resultTags = result.getTags();
```

Managing tags using the AWS CLI

The following sections show how to use the AWS CLI to create and manage tags on data catalogs.

Adding tags to a resource: Tag-resource

The `tag-resource` command adds one or more tags to a specified resource.

**Syntax**

```bash
aws athena tag-resource --resource-arn
arn:aws:athena:region:account_id:datacatalog/catalog_name --tags
Key=string,Value=string  Key=string,Value=string
```

The `--resource-arn` parameter specifies the resource to which the tags are added. The `--tags` parameter specifies a list of space-separated key-value pairs to add as tags to the resource.
Example
The following example adds tags to the mydatacatalog data catalog.

```
```

To show the result, use the list-tags-for-resource command.

For information about adding tags when using the create-data-catalog command, see Registering a catalog: Create-data-catalog (p. 54).

Listing the tags for a resource: List-tags-for-resource

The list-tags-for-resource command lists the tags for the specified resource.

Syntax

```
aws athena list-tags-for-resource --resource-arn arn:aws:athena:region:account_id:datacatalog/catalog_name
```

The --resource-arn parameter specifies the resource for which the tags are listed.

The following example lists the tags for the mydatacatalog data catalog.

```
aws athena list-tags-for-resource --resource-arn arn:aws:athena:us-east-1:111122223333:datacatalog/mydatacatalog
```

The following sample result is in JSON format.

```
{
  "Tags": [
    {
      "Key": "Time",
      "Value": "Now"
    },
    {
      "Key": "Color",
      "Value": "Orange"
    }
  ]
}
```

Removing tags from a resource: Untag-resource

The untag-resource command removes the specified tag keys and their associated values from the specified resource.

Syntax

```
aws athena untag-resource --resource-arn arn:aws:athena:region:account_id:datacatalog/catalog_name --tag-keys key_name [key_name ...]
```

The --resource-arn parameter specifies the resource from which the tags are removed. The --tag-keys parameter takes a space-separated list of key names. For each key name specified, the untag-resource command removes both the key and its value.

The following example removes the Color and Time keys and their values from the mydatacatalog catalog resource.
Tag-based IAM access control policies

Having tags allows you to write an IAM policy that includes the Condition block to control access to a resource based on its tags.

Tag policy examples for workgroups

Example 1. basic tagging policy

The following IAM policy allows you to run queries and interact with tags for the workgroup named workgroupA:

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "athena:ListWorkGroups",
                "athena:ListEngineVersions",
                "athena:ListDataCatalogs",
                "athena:ListDatabases",
                "athena:GetDatabase",
                "athena:ListTableMetadata",
                "athena:GetTableMetadata"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "athena:GetWorkGroup",
                "athena:TagResource",
                "athena:UntagResource",
                "athena:ListTagsForResource",
                "athena:StartQueryExecution",
                "athena:GetQueryExecution",
                "athena:BatchGetQueryExecution",
                "athena:ListQueryExecutions",
                "athena:StopQueryExecution",
                "athena:GetQueryResults",
                "athena:GetQueryResultsStream",
                "athena:CreateNamedQuery",
                "athena:GetNamedQuery",
                "athena:BatchGetNamedQuery",
                "athena:ListNamedQueries",
                "athena:DeleteNamedQuery",
                "athena:CreatePreparedStatement",
                "athena:GetPreparedStatement",
                "athena:ListPreparedStatements",
                "athena:UpdatePreparedStatement",
                "athena:DeletePreparedStatement"
            ],
        }
    ]
}
```
Example 2: Policy block that denies actions on a workgroup based on a tag key and tag value pair

Tags that are associated with a resource like a workgroup are referred to as resource tags. Resource tags let you write policy blocks like the following that deny the listed actions on any workgroup tagged with a key-value pair like stack, production.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "athena:GetWorkGroup",
                "athena:UpdateWorkGroup",
                "athena:DeleteWorkGroup",
                "athena:TagResource",
                "athena:UntagResource",
                "athena:ListTagsForResource",
                "athena:StartQueryExecution",
                "athena:GetQueryExecution",
                "athena:BatchGetQueryExecution",
                "athena:ListQueryExecutions",
                "athena:StopQueryExecution",
                "athena:GetQueryResults",
                "athena:GetQueryResultsStream",
                "athena:CreateNamedQuery",
                "athena:GetNamedQuery",
                "athena:BatchGetNamedQuery",
                "athena:ListNamedQueries",
                "athena:DeleteNamedQuery",
                "athena:CreatePreparedStatement",
                "athena:GetPreparedStatement",
                "athena:ListPreparedStatements",
                "athena:UpdatePreparedStatement",
                "athena:DeletePreparedStatement"
            ],
            "Resource": "arn:aws:athena:us-east-1:123456789012:workgroup/*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/stack": "production"
                }
            }
        }
    ]
}
```

Example 3. Policy block that restricts tag-changing action requests to specified tags

Tags that are passed in as parameters to operations that change tags (for example, TagResource, UntagResource, or CreateWorkGroup with tags) are referred to as request tags. The following example policy block allows the CreateWorkGroup operation only if one of the tags passed has the key costcenter and the value 1, 2, or 3.

**Note**

If you want to allow IAM users to pass in tags as part of a CreateWorkGroup operation, make sure that you give the users permissions to the TagResource and CreateWorkGroup actions.

```
{
    "Version": "2012-10-17",
}
```
Tag policy examples for data catalogs

Example 1. basic tagging policy

The following IAM policy allows you to interact with tags for the data catalog named datacatalogA:

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "athena:ListWorkGroups",
                "athena:ListEngineVersions",
                "athena:ListDataCatalogs",
                "athena:ListDatabases",
                "athena:GetDatabase",
                "athena:ListTableMetadata",
                "athena:GetTableMetadata"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "athena:GetWorkGroup",
                "athena:TagResource",
                "athena:UntagResource",
                "athena:ListTagsForResource",
                "athena:StartQueryExecution",
                "athena:GetQueryExecution",
                "athena:BatchGetQueryExecution",
                "athena:ListQueryExecutions",
                "athena:StopQueryExecution",
                "athena:GetQueryResults",
                "athena:GetQueryResultsStream",
                "athena:CreateNamedQuery",
                "athena:GetNamedQuery",
                "athena:BatchGetNamedQuery",
                "athena:ListNamedQueries",
                "athena:DeleteNamedQuery"
            ],
            "Resource": "*"
        }
    ]
}
```
Example 2: Policy block that denies actions on a Data Catalog based on a tag key and tag value pair

You can use resource tags to write policy blocks that deny specific actions on data catalogs that are tagged with specific tag key-value pairs. The following example policy denies actions on data catalogs that have the tag key-value pair stack, production.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "athena:CreateDataCatalog",
                "athena:GetDataCatalog",
                "athena:UpdateDataCatalog",
                "athena:DeleteDataCatalog",
                "athena:GetDatabase",
                "athena:ListDatabases",
                "athena:GetTableMetadata",
                "athena:GetTableMetadata",
                "athena:StartQueryExecution",
                "athena:TagResource",
                "athena:UntagResource",
                "athena:ListTagsForResource"
            ],
        }
    ],
    "Resource": "arn:aws:athena:us-east-1:123456789012:datacatalog/*",
    "Condition": {
        "StringEquals": {
            "aws:ResourceTag/stack": "production"
        }
    }
}
```
Example 3. policy block that restricts tag-changing action requests to specified tags

Tags that are passed in as parameters to operations that change tags (for example, TagResource, UntagResource, or CreateDataCatalog with tags) are referred to as request tags. The following example policy block allows the CreateDataCatalog operation only if one of the tags passed has the key costcenter and the value 1, 2, or 3.

Note
If you want to allow IAM users to pass in tags as part of a CreateDataCatalog operation, make sure that you give the users permissions to the TagResource and CreateDataCatalog actions.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "athena:CreateDataCatalog",
        "athena:TagResource"
      ],
      "Resource": "arn:aws:athena:us-east-1:123456789012:datacatalog/*",
      "Condition": {
        "StringEquals": {
          "aws:RequestTag/costcenter": [
            "1",
            "2",
            "3"
          ]
        }
      }
    }
  ]
}
```
Athena engine versioning

Athena occasionally releases a new engine version to provide improved performance, functionality, and code fixes. When a new engine version is available, Athena notifies you through the Athena console and your AWS Health Dashboard. Your AWS Health Dashboard notifies you about events that can affect your AWS services or account. For more information about AWS Health Dashboard, see Getting started with the AWS Health Dashboard.

Engine versioning is configured per workgroup (p. 629). You can use workgroups to control which query engine your queries use. The query engine that is in use is shown in the query editor, on the details page for the workgroup, and by the Athena APIs.

You can choose to upgrade your workgroups as soon as a new engine is available or continue using the older version until it is no longer supported. You can also let Athena decide when to upgrade your workgroups. This is the default setting. If you take no action, Athena notifies you in advance prior to upgrading your workgroups. If you let Athena decide, Athena upgrades your workgroups for you unless it finds incompatibilities.

When you start using a new engine version, a small subset of queries may break due to incompatibilities. You can use workgroups to test your queries in advance of the upgrade by creating a test workgroup that uses the new engine or by test upgrading an existing workgroup. For more information, see Testing queries in advance of an engine version upgrade (p. 671).

Topics
- Changing Athena engine versions (p. 669)
- Athena engine version reference (p. 672)

Changing Athena engine versions

Athena occasionally releases a new engine version to provide improved performance, functionality, and code fixes. When a new engine version is available, Athena notifies you in the console. You can choose to let Athena decide when to upgrade, or manually specify an Athena engine version per workgroup.

Topics
- Finding the query engine version for a workgroup (p. 669)
- Changing the engine version using the Athena console (p. 670)
- Changing the engine version using the AWS CLI (p. 671)
- Specifying the engine version when you create a workgroup (p. 671)
- Testing queries in advance of an engine version upgrade (p. 671)
- Troubleshooting queries that fail (p. 672)

Finding the query engine version for a workgroup

You can use the Workgroups page to find the current engine version for any workgroup.
Changing the engine version using the Athena console

When a new engine version is available, you can choose to let Athena decide when to upgrade the workgroup, or manually specify the Athena engine version that the workgroup uses. If only one version is currently available, manually specifying a different version is not possible.

**Note**
To change the engine version for a workgroup, you must have permission to perform the athena:ListEngineVersions action on the workgroup. For IAM policy examples, see Workgroup example policies (p. 633).

**To let Athena decide when to upgrade the workgroup**

2. If the console navigation pane is not visible, choose the expansion menu on the left.
3. In the console navigation pane, choose Workgroups.
4. In the list of workgroups, choose the link for the workgroup that you want to configure.
5. Choose Edit.
6. In the Query engine version section, for Update query engine, choose Automatic to let Athena choose when to upgrade your workgroup. This is the default setting.
7. Choose Save changes.

In the list of workgroups, the Query engine update status for the workgroup shows Automatic.

**To manually choose an engine version**

2. If the console navigation pane is not visible, choose the expansion menu on the left.
3. In the console navigation pane, choose Workgroups.
4. In the list of workgroups, choose the link for the workgroup that you want to configure.
5. Choose Edit.
6. In the Query engine version section, for Update query engine, choose Manual to manually choose an engine version.
7. Use the Query engine version option to choose the engine version that you want the workgroup to use. If a different engine version is unavailable, a different engine version cannot be specified.
8. Choose Save changes.

In the list of workgroups, the Query engine update status for the workgroup shows Manual.

Changing the engine version using the AWS CLI

To change the engine version using the AWS CLI, use the syntax in the following example.

```
aws athena update-work-group --work-group workgroup-name --configuration-updates
EngineVersion={SelectedEngineVersion='Athena engine version 3'}
```

Specifying the engine version when you create a workgroup

When you create a workgroup, you can specify the engine version that the workgroup uses or let Athena decide when to upgrade the workgroup. If a new engine version is available, a best practice is to create a workgroup to test the new engine before you upgrade your other workgroups. To specify the engine version for a workgroup, you must have the athena:ListEngineVersions permission on the workgroup. For IAM policy examples, see Workgroup example policies (p. 633).

To specify the engine version when you create a workgroup

2. If the console navigation pane is not visible, choose the expansion menu on the left.
3. In the console navigation pane, choose Workgroups.
4. On the Workgroups page, choose Create workgroup.
5. On the Create workgroup page, in the Query engine version section, do one of the following:
   - Choose Automatic to let Athena choose when to upgrade your workgroup. This is the default setting.
   - Choose Manual to manually choose a different engine version if one is available.
6. Enter information for the other fields as necessary. For information about the other fields, see Create a workgroup (p. 640).
7. Choose Create workgroup.

Testing queries in advance of an engine version upgrade

When a workgroup is upgraded to a new engine version, some of your queries can break due to incompatibilities. To make sure that your engine version upgrade goes smoothly, you can test your queries in advance.
To test your queries prior to an engine version upgrade

1. Verify the engine version of the workgroup that you are using. The engine version that you are using is displayed on the Workgroups page in the Query engine version column for the workgroup. For more information, see Finding the query engine version for a workgroup (p. 669).
2. Create a test workgroup that uses the new engine version. For more information, see Specifying the engine version when you create a workgroup (p. 671).
3. Use the new workgroup to run the queries that you want to test.
4. If a query fails, use the Athena engine version reference (p. 672) to check for breaking changes that might be affecting the query. Some changes may require you to update the syntax of your queries.
5. If your queries still fail, contact AWS Support for assistance. In the AWS Management Console, choose Support, Support Center, or ask a question on AWS re:Post using the Amazon Athena tag.

Troubleshooting queries that fail

If a query fails after an engine version upgrade, use the Athena engine version reference (p. 672) to check for breaking changes, including changes that may affect the syntax in your queries.

If your queries still fail, contact AWS Support for assistance. In the AWS Management Console, choose Support, Support Center, or ask a question on AWS re:Post using the Amazon Athena tag.

Athena engine version reference

This section lists the changes to the Athena query engine.

Topics

- Athena engine version 3 (p. 672)
- Athena engine version 2 (p. 680)

Athena engine version 3

For engine version 3, Athena has introduced a continuous integration approach to open source software management that improves currency with the Trino and Presto projects so that you get faster access to community improvements, integrated and tuned within the Athena engine.

This release of Athena engine version 3 supports all the features of Athena engine version 2. This document highlights key differences between Athena engine version 2 and Athena engine version 3.

- Getting started (p. 672)
- Improvements and new features (p. 673)
- Limitations (p. 676)
- Breaking changes (p. 677)

Getting started

To get started, either create a new Athena workgroup that uses Athena engine version 3 or configure an existing workgroup to use version 3. Any Athena workgroup can upgrade from engine version 2 to engine version 3 without interruption in your ability to submit queries.

For more information, see Changing Athena engine versions.
Improvements and new features

The features and updates listed include improvements from Athena itself and from functionality incorporated from open source Trino. For an exhaustive list of SQL query operators and functions, refer to the Trino documentation.

Added Features

Apache Spark bucketing algorithm support

Athena can read buckets generated by the Spark hash algorithm. To specify that data was originally written by the Spark hash algorithm, put (`'bucketing_format'='spark'`) in the TBLPROPERTIES clause of your CREATE TABLE statement. If this property is not specified, the Hive hash algorithm is used.

```
CREATE EXTERNAL TABLE `spark_bucket_table`(
  `id` int,
  `name` string
) CLUSTERED BY (`name`) INTO 8 BUCKETS STORED AS PARQUET LOCATION 's3://path/to/bucketed/table/' TBLPROPERTIES ('bucketing_format'='spark')
```

Added Functions

The functions in this section are new to Athena engine version 3.

**Aggregate functions**

`listagg(x, separator)` – Returns the concatenated input values, separated by the separator string.

```
SELECT listagg(value, ',') WITHIN GROUP (ORDER BY value) csv_value
FROM (VALUES 'a', 'c', 'b') t(value);
```

**Array functions**

`contains_sequence(x, seq)` – Returns true if array x contains all array seq as a sequential subset (all values in the same consecutive order).

```
SELECT contains_sequence(ARRAY [1,2,3,4,5,6], ARRAY[1,2]);
```

**Binary functions**

`murmur3(binary)` – Computes the 128-bit MurmurHash3 hash of binary.

```
SELECT murmur3(from_base64('aaaaaa'));
```

**Conversion functions**

`format_number(number)` – Returns a formatted string using a unit symbol.

```
SELECT format_number(123456); -- '123K'
SELECT format_number(1000000); -- '1M'
```
Date and time functions

timezone_hour(timestamp) – Returns the hour of the time zone offset from timestamp.

```sql
SELECT EXTRACT(TIMEZONE_HOUR FROM TIMESTAMP '2020-05-10 12:34:56 +08:35');
```

timezone_minute(timestamp) – Returns the minute of the time zone offset from timestamp.

```sql
SELECT EXTRACT(TIMEZONE_MINUTE FROM TIMESTAMP '2020-05-10 12:34:56 +08:35');
```

Geospatial functions

to_encoded_polyline(Geometry) – Encodes a linestring or multipoint to a polyline.

```sql
SELECT to_encoded_polyline(ST_GeometryFromText(  'LINESTRING (-120.2 38.5, -120.95 40.7, -126.453 43.252)'));
```

from_encoded_polyline(varchar) – Decodes a polyline to a linestring.

```sql
SELECT ST_AsText(from_encoded_polyline('_p~iF~ps|U_ulLnnqC_mqNvxq`@'));
```

to_geojson_geometry(SphericalGeography) – Returns the specified spherical geography in GeoJSON format.

```sql
SELECT to_geojson_geometry(to_spherical_geography(ST_GeometryFromText(  'LINESTRING (0 0, 1 2, 3 4)')));
```

from_geojson_geometry(varchar) – Returns the spherical geography type object from the GeoJSON representation, stripping non geometry key/values. Feature and FeatureCollection are not supported.

```sql
SELECT from_geojson_geometry(to_geojson_geometry(to_spherical_geography(ST_GeometryFromText(  'LINESTRING (0 0, 1 2, 3 4)'))));
```

ground nearest points(Geometry, Geometry) – Returns the points on each geometry that are nearest each other. If either geometry is empty, returns NULL. Otherwise, returns a row of two Point objects that have the minimum distance of any two points on the geometries. The first point is from the first Geometry argument, the second from the second Geometry argument. If there are multiple pairs with the same minimum distance, one pair is chosen arbitrarily.

```sql
SELECT geometry_nearest_points(ST_GeometryFromText(  'LINESTRING (50 100, 50 200)'), ST_GeometryFromText(  'LINESTRING (10 10, 20 20)'));
```

Set Digest functions

make_set_digest(x) – Composes all input values of x into a setdigest.

```sql
SELECT make_set_digest(value) FROM (VALUES 1, 2, 3) T(value);
```

String functions

soundex(char) – Returns a character string that contains the phonetic representation of char.

```sql
SELECT soundex(value) FROM (VALUES 'hello', 'bello') T(value);
```
SELECT name
FROM nation
WHERE SOUNDEx(name) = SOUNDEx('CHYNA'); -- CHINA

concat_ws(string0, string1, ..., stringN) – Returns the concatenation of string1, string2, ..., stringN using string0 as a separator. If string0 is null, then the return value is null. Any null values provided in the arguments after the separator are skipped.

SELECT concat_ws(',', 'def', 'pqr', 'mno');

System functions

version() – Returns the server version.

SELECT version();

Window functions

GROUPS – Adds support for window frames based on groups.

SELECT array_agg(a) OVER(
    ORDER BY a ASC NULLS FIRST GROUPS BETWEEN 1 PRECEDING AND 2 FOLLOWING)
FROM (VALUES 3, 3, 3, 2, 2, 1, null, null) T(a);

Performance improvements

Performance improvements in Athena engine version 3 include the following.

- Faster AWS Glue table metadata retrieval – Queries that involve multiple tables will see reduced query planning time.
- Dynamic filtering for RIGHT JOIN – Dynamic filtering is now enabled for right joins that have equality join conditions, as in the following example.

SELECT *
FROM lineitem RIGHT JOIN tpch.tiny.supplier
ON lineitem.suppkey = supplier.suppkey
WHERE supplier.name = 'abc';

Reliability enhancements

General engine memory usage and tracking in Athena engine version 3 have been improved. Large queries are less susceptible to failure from node crashes.

Query syntax enhancements

INTERSECT ALL – Added support for INTERSECT ALL.

SELECT * FROM (VALUES 1, 2, 3, 4) INTERSECT ALL SELECT * FROM (VALUES 3, 4);

EXCEPT ALL – Added support for EXCEPT ALL.

SELECT * FROM (VALUES 1, 2, 3, 4) EXCEPT ALL SELECT * FROM (VALUES 3, 4);

RANGE PRECEDING – Added support for RANGE PRECEDING in window functions.
**SELECT** sum(x) over (order by x range 1 preceding) 
**FROM** (values (1), (1), (2), (2)) t(x);

**MATCH_RECOGNIZE** – Added support for row pattern matching, as in the following example.

```sql
SELECT m.id AS row_id, m.match, m.val, m.label
FROM (VALUES(1, 90),(2, 80),(3, 70),(4, 70)) t(id, value)
MATCH_RECOGNIZE ( 
    ORDER BY id
    MEASURES match_number() AS match,
    RUNNING LAST(value) AS val,
    classifier() AS label
    ALL ROWS PER MATCH
    AFTER MATCH SKIP PAST LAST ROW
    PATTERN ((() | A) DEFINE A AS true
) AS m;
```

**Data format and data type enhancements**

Athena engine version 3 has the following data format and data type enhancements.

- **LZ4 and ZSTD** – Added support for reading LZ4 and ZSTD compressed Parquet data. Added support for writing ZSTD compressed ORC data.
- **Symlink-based tables** – Added support for creating symlink-based tables on Avro files. An example follows.

```sql
CREATE TABLE test_avro_symlink
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
... 
INPUTFORMAT 'org.apache.hadoop.hive.ql.io.SymlinkTextInputFormat'
```

- **SphericalGeography** – The SphericalGeography type provides native support for spatial features represented on geographic coordinates (sometimes called geodetic coordinates, lat/lon, or lon/lat). Geographic coordinates are spherical coordinates expressed in angular units (degrees).

The `to_spherical_geography` function returns geographic (spherical) coordinates from geometric (planar) coordinates, as in the following example.

```sql
SELECT to_spherical_geography(ST_GeometryFromText('LINESTRING (-40.2 28.9, -40.2 31.9, -37.2 31.9)'));
```

**Miscellaneous enhancements**

- **Large prepared statements** – Increased the default HTTP request/response header size to 2MB to allow large prepared statements.

**Limitations**

Athena engine version 3 has the following limitations.

- **Query performance** – Many queries run faster on Athena engine version 3, but some query plans can differ from Athena engine version 2. As a result, some queries can differ in latency or cost.
- **Trino and Presto connectors** – Neither Trino nor Presto connectors are supported. Use Amazon Athena Federated Query to connect data sources. For more information, see Using Amazon Athena Federated Query (p. 60).
• **Fault-tolerant execution** – Trino fault-tolerant execution (Trino Tardigrade) is not supported.

## Breaking changes

When you migrate from Athena engine version 2 to Athena engine version 3, certain changes can affect table schema, syntax, or data type usage. This section lists the associated error messages and provides suggested workarounds.

### Query syntax changes

**Minute() function does not support interval year to month**

**Error message:** Unexpected parameters (interval year to month) for function minute. Expected: minute(timestamp with time zone), minute(time with time zone), minute(timestamp), minute(time), minute(interval day to second).

**Cause:** In Athena engine version 3, type checks have been made more precise for EXTRACT in accordance with the ANSI SQL specification.

**Suggested solution:** Update the queries to make sure types are matched with the suggested function signatures.

**BOOL_OR() and BOOL_AND() cannot be used together with null treatment**

**Error message:** Cannot specify null treatment clause for bool_or function.

**Cause:** Ignore nulls cannot be added for the BOOL_OR() function. This change was made to conform to the ANSI SQL specification.

**Suggested solution:** Remove ignore nulls from the query strings.

**ORDER BY expressions must appear in SELECT list**

**Error message:** For SELECT DISTINCT, ORDER BY expressions must appear in SELECT list

**Cause:** Incorrect table aliasing is used in a SELECT clause.

**Suggested solution:** Double check that all columns in the ORDER BY expression have proper references in the SELECT DISTINCT clause.

**Geospatial function does not support varbinary input**

**Error message:** FUNCTION_NOT_FOUND for st_XXX

**Cause:** A few geospatial functions no longer support the legacy VARBINARY input type or text related function signatures.

**Suggested solution:** Use geospatial functions to convert the input types to types that are supported. Supported input types are indicated in the error message.

**Zero-length delimited identifier not allowed**

**Error message:** Zero-length delimited identifier not allowed.

**Cause:** A query used an empty string as a column alias.

**Suggested solution:** Update the query to use a non-empty alias for the column.

**Columns in a SELECT clause must exist**

**Error message:** Mismatched input XXX. Expecting: YYY
**Cause**: The column projected does not exist in any of the subqueries.

**Suggested solution**: Update the query to remove the nonexistent columns.

**SYSTEM_TIME and SYSTEM_VERSION clauses deprecated for time travel**

**Error message**: mismatched input 'SYSTEM_TIME'. Expecting: 'TIMESTAMP', 'VERSION'

**Cause**: In Athena engine version 2, Iceberg and Lake Formation governed tables used the `FOR SYSTEM_TIME AS OF` and `FOR SYSTEM_VERSION AS OF` clauses for timestamp and version time travel. Athena engine version 3 uses the `FOR TIMESTAMP AS OF` and `FOR VERSION AS OF` clauses.

**Suggested solution**: Update the SQL query to use the `TIMESTAMP AS OF` and `VERSION AS OF` clauses for time travel operations, as in the following examples.

Time travel by timestamp:

```
SELECT * FROM TABLE FOR TIMESTAMP AS OF (current_timestamp - interval '1' day)
```

Time travel by version:

```
SELECT * FROM TABLE FOR VERSION AS OF 949530903748831860
```

**The CONCAT function must have two or more arguments**

**Error Message**: INVALID_FUNCTION_ARGUMENT: There must be two or more concatenation arguments

**Cause**: Previously, the `CONCAT` string function accepted a single argument. In Athena engine version 3, the `CONCAT` function requires a minimum of two arguments.

**Suggested solution**: Change occurrences of `CONCAT(str)` to `CONCAT(str, '')`.

**Data processing changes**

**Unsupported decimal column type coercion**

**Error message**: Unsupported column type (varchar) for Parquet column ([XXX] optional binary column

**Cause**: When doing conversions, Athena engine version 2 read values at the byte level. Athena engine version 3 introduces type validation and checks that the type is compatible before trying to read the value.

**Suggested Solution**: Update your AWS Glue schema and make sure that decimal type columns in Parquet files are not defined as VARCHAR in AWS Glue.

**Substrings from character arrays no longer contain padded spaces**

**Error message**: No error is thrown, but the string returned no longer contains padded spaces. For example, `substr(char[20],1,100)` now returns a string with length 20 instead of 100.

**Suggested solution**: No action is required.

**Nulls in List data types are now propagated to UDFs**

**Error message**: Null Pointer Exception

**Cause**: This issue can affect you if you use the UDF connector and have implemented a user defined Lambda function.
Athena engine version 2 filtered out the nulls in List data types that were passed to a user defined function. In Athena engine version 3, the nulls are now preserved and passed on to the UDF. This can cause a null pointer exception if the UDF attempts to dereference the null element without checking.

For example, if you have the data `[null, 1, null, 2, 3, 4]` in an originating data source like DynamoDB, the following are passed to the user-defined Lambda function:

**Athena engine version 2**: `[1, 2, 3, 4]`

**Athena engine version 3**: `[null, 1, null, 2, 3, 4]`

**Suggested solution**: Ensure that your user-defined Lambda function handles null elements in list data types.

**Iceberg table column level security enforcement change**

**Error Message**: Access Denied: Cannot select from columns

**Cause**: The Iceberg table was created outside Athena and uses an Apache Iceberg SDK version earlier than 0.13.0. Because earlier SDK versions do not populate columns in AWS Glue, Lake Formation could not determine the columns authorized for access.

**Suggested solution**: Perform an update using the Athena ALTER TABLE SET PROPERTIES (p. 508) statement or use the latest Iceberg SDK to fix the table and update the column information in AWS Glue.

**Timestamp changes**

**Date timestamp overflow throws error**

**Error message**: Millis overflow: XXX

**Cause**: Because ISO 8601 dates were not checked for overflow in Athena engine version 2, some dates produced a negative timestamp. Athena engine version 3 checks for this overflow and throws an exception.

**Suggested Solution**: Make sure the timestamp is within range.

**Timestamp overflow for int96 Parquet format**

**Error message**: Invalid timeOfDayNanos

**Cause**: A timestamp overflow for the int96 Parquet format.

**Suggested solution**: Identify the specific files that have the issue. Then generate the data file again with an up-to-date, well known Parquet library, or use Athena CTAS. If the issue persists, contact Athena support and let us know how the data files are generated.

**Casting from TimestampTZ to Timestamp is not supported**

**Error message**: Casting a Timestamp with Time Zone to Timestamp is not supported.

**Suggested solution**: Any explicit or implicit cast from TimestampTZ to Timestamp throws the exception. If possible, remove the cast and use a different data type.

**Precision mismatch in Timestamp columns causes serialization error**

**Error message**: SERIALIZATION_ERROR: Could not serialize column 'COLUMNZ' of type 'timestamp(3)' at position X:Y

 COLUMNZ is the output name of the column that causes the issue. The numbers X:Y indicate the position of the column in the output.
**Cause:** Athena engine version 3 checks to make sure that the precision of timestamps in the data is the same as the precision specified for the column data type in the table specification. Currently, this precision is always 3. If the data has a precision greater than this, queries fail with the error noted.

**Suggested solution:** Check your data to make sure that your timestamps have millisecond precision.

**Timestamp values with precision greater than 3 are rounded**

**Error message:** There is no error message, but values that were trimmed in Athena engine version 2 are rounded in Athena engine version 3.

**Suggested solution:** Exercise care when using Timestamp values with a precision greater than 3.

**Implicit type conversion from Timestamp type to the Int or BigInt type is not supported**

**Error message:** GENERIC_INTERNAL_ERROR: class io.trino.spi.type.BigIntType cannot be cast to class io.trino.spi.type.TimestampType

**Cause:** A type mismatch between a Parquet column type (Timestamp) and a schema column type (Int or BigInt).

**Suggested solution:** Change a Parquet column type or a schema column type to have the same date type. Because changing a Parquet column type requires a data update, changing a schema column type is preferable.

**Political time zones with TIME not supported**

**Error message:** INVALID LITERAL

**Cause:** Queries like `SELECT TIME '13:21:32.424 America/Los_Angeles'`.

**Suggested solution:** Avoid using political time zones with TIME.

**Time and interval year to month not supported**

**Error message:** TYPE MISMATCH

**Cause:** Athena engine version 3 does not support time and interval year to month (for example, `SELECT TIME '01:00' + INTERVAL '3' MONTH`).

**Casting a Timestamp with time zone to varchar behavior change**

In Athena engine version 2, casting a Timestamp with time zone to varchar caused some time zone literals to change (for example, US/Eastern changed to America/New_York). This behavior does not occur in Athena engine version 3.

**Reading the Long type as Timestamp in ORC files now causes a malformed ORC file error**

**Error message:** Error opening Hive split 'FILE (SPLIT POSITION)' Malformed ORC file. Cannot read SQL type timestamp from ORC stream .long_type of type LONG

**Cause:** Athena engine version 3 now rejects implicit coercion from the Long data type to Timestamp. Previously, Long values were implicitly converted into timestamp as if they were epoch milliseconds.

**Suggested solution:** Use the `from_unixtime` function to explicitly cast the column, or use the `from_unixtime` function to create an additional column for future queries.

### Athena engine version 2

Athena engine version 2 introduces the following changes. Currently, Athena engine version 2 is available in all Regions where Athena is supported, including Africa (Cape Town), Asia Pacific (Hong Kong), Asia Pacific (Mumbai), Asia Pacific (Osaka), Asia Pacific (Seoul), Asia Pacific (Singapore), Asia Pacific (Sydney),
Asia Pacific (Tokyo), AWS GovCloud (US-East), AWS GovCloud (US-West), Canada (Central), China (Beijing), China (Ningxia), Europe (Frankfurt), Europe (Ireland), Europe (London), Europe (Milan), Europe (Paris), Europe (Stockholm), Middle East (Bahrain), South America (São Paulo), US East (N. Virginia), US East (Ohio), US West (N. California), and US West (Oregon).

- Improvements and new features (p. 681)
  - Grouping, join, and subquery improvements (p. 681)
  - Datatype enhancements (p. 682)
  - Added functions (p. 683)
  - Performance improvements (p. 684)
  - JSON-related improvements (p. 686)
- Breaking changes (p. 686)
  - Bug fixes (p. 686)
  - Changes to geospatial functions (p. 687)
  - ANSI SQL compliance (p. 688)
  - Replaced functions (p. 688)
  - Limits (p. 688)

### Improvements and new features

- **EXPLAIN** and **EXPLAIN ANALYZE** – You can use the EXPLAIN statement in Athena to view the execution plan for your SQL queries. Use EXPLAIN ANALYZE to view the distributed execution plan for your SQL queries and the cost of each operation. For more information, see Using EXPLAIN and EXPLAIN ANALYZE in Athena (p. 703).
- **Federated queries** – Federated queries are supported in Athena engine version 2. For more information, see Using Amazon Athena Federated Query (p. 60).
- **Geospatial functions** – More than 25 geospatial functions have been added. For more information, see New geospatial functions in Athena engine version 2 (p. 417).
- **Nested schema** – Support has been added for reading nested schema, which reduces cost.
- **Prepared statements** – Use prepared statements for repeated execution of the same query with different query parameters. A prepared statement contains placeholder parameters whose values you pass at runtime. Prepared statements help prevent SQL injection attacks. For more information, see Using parameterized queries (p. 370).
- **Schema evolution support** – Schema evolution support has been added for data in Parquet format.
  - Added support for reading array, map, or row type columns from partitions where the partition schema is different from the table schema. This can occur when the table schema was updated after the partition was created. The changed column types must be compatible. For row types, trailing fields may be added or dropped, but the corresponding fields (by ordinal) must have the same name.
  - ORC files can now have struct columns with missing fields. This allows the table schema to be changed without rewriting the ORC files.
  - ORC struct columns are now mapped by name rather than ordinal. This correctly handles missing or extra struct fields in the ORC file.
- **SQL OFFSET** – The SQL OFFSET clause is now supported in SELECT statements. For more information, see SELECT (p. 692).
- **UNLOAD statement** – You can use the UNLOAD statement to write the output of a SELECT query to the PARQUET, ORC, AVRO, and JSON formats. For more information, see UNLOAD (p. 717).

### Grouping, join, and subquery improvements

- **Complex grouping** – Added support for complex grouping operations.
• **Correlated subqueries** – Added support for correlated subqueries in IN predicates and for correlated subqueries that require coercions.

• **CROSS JOIN** – Added support for CROSS JOIN against LATERAL derived tables.

• **GROUPING SETS** – Added support for ORDER BY clauses in aggregations for queries that use GROUPING SETS.

• **Lambda expressions** – Added support for dereferencing row fields in Lambda expressions.

• **Null values in semijoins** – Added support for null values on the left-hand side of a semijoin (that is, an IN predicate with subqueries).

• **Spatial joins** – Added support for broadcast spatial joins and spatial left joins.

• **Spill to disk** – For memory intensive INNER JOIN and LEFT JOIN operations, Athena offloads intermediate operation results to disk. This enables execution of queries that require large amounts of memory.

**Datatype enhancements**

• **INT for INTEGER** – Added support for INT as an alias for the INTEGER data type.

• **INTERVAL types** – Added support for casting to INTERVAL types.

• **IPADDRESS** – Added a new IPADDRESS type to represent IP addresses. Added support for casting between the VARBINARY type and IPADDRESS type.

• **IS DISTINCT FROM** – Added IS DISTINCT FROM support for the JSON and IPADDRESS types.

• **Null equality checks** – Equality checks for null values in ARRAY, MAP, and ROW data structures are now supported. For example, the expression ARRAY ['1', '3', null] = ARRAY ['1', '2', null] returns false. Previously, a null element returned the error message comparison not supported.

• **Row type coercion** – Coercion between row types regardless of field names is now allowed. Previously, a row type was coercible to another only if the field name in the source type matched the target type, or when the target type had an anonymous field name.

• **Time subtraction** – Implemented subtraction for all TIME and TIMESTAMP types.

• **Unicode** – Added support for escaped Unicode sequences in string literals.

• **VARBINARY concatenation** – Added support for concatenation of VARBINARY values.

**Window value functions** – Window value functions now support IGNORE NULLS and RESPECT NULLS.

**Additional input types for functions**

The following functions now accept additional input types. For more information about each function, visit the corresponding link to the Presto documentation.

• **approx_distinct()** – The approx_distinct() function now supports the following types: INTEGER, SMALLINT, TINYINT, DECIMAL, REAL, DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIME, TIME WITH TIME ZONE, IPADDRESS, and CHAR.

• **Avg(), sum()** – The avg() and sum() aggregate functions now support the INTERVAL data type.

• **Lpad(), rpad()** – The lpad and rpad functions now work on VARBINARY inputs.

• **Min(), max()** – The min and max aggregation functions now allow unknown input types at query analysis time so that you can use the functions with NULL literals.

• **regexp_replace()** – Variant of the regexp_replace() function added that can execute a Lambda function for each replacement.

• **Sequence()** – Added DATE variants for the sequence() function, including variant with an implicit one-day step increment.

• **ST_Area()** – The ST_Area() geospatial function now supports all geometry types.
• **Substr()** – The `substr` function now works on `VARBINARY` inputs.

• **zip_with()** – Arrays of mismatched length can now be used with `zip_with()`. Missing positions are filled with null. Previously, an error was raised when arrays of differing lengths were passed. This change may make it difficult to distinguish between values that were originally null from values that were added to pad the arrays to the same length.

**Added functions**

The following list contains functions that are new starting in Athena engine version 2. The list does not include geospatial functions. For a list of geospatial functions, see New geospatial functions in Athena engine version 2 (p. 417).

For more information about each function, visit the corresponding link to the Presto documentation.

**Aggregate functions**

`reduce_agg()`

**Array functions and operators**

`array_sort()` - Variant of this function added that takes a Lambda function as a comparator.

`ngrams()`

**Binary functions and operators**

`from_big_endian_32()`

`from_ieee754_32()`

`from_ieee754_64()`

`hmac_md5()`

`hmac_sha1()`

`hmac_sha256()`

`hmac_sha512()`

`spooky_hash_v2_32()`

`spooky_hash_v2_64()`

`to_big_endian_32()`

`to_ieee754_32()`

`to_ieee754_64()`

**Date and time functions and operators**

`millisecond()`

`parse_duration()`

`to_milliseconds()`

**Map functions and operators**

`multimap_from_entries()`
Mathematical functions and operators

inverse_normal_cdf()
 wilson_interval_lower()
 wilson_interval_upper()

Quantile digest functions

quantile digest functions and the qdigest quantile digest type added.

String functions and operators

hamming_distance()
 split_to_multimap()

Performance improvements

Performance of the following features has improved in Athena engine version 2.

Query performance

- **Broadcast join performance** – Improved broadcast join performance by applying dynamic partition pruning in the worker node.
- **Bucketed tables** – Improved performance for writing to bucketed tables when the data being written is already partitioned appropriately (for example, when the output is from a bucketed join).
- **DISTINCT** – Improved performance for some queries that use DISTINCT.

  Dynamic filtering and partition pruning – Improvements increase performance and reduce the amount of data scanned in queries.
- **Filter and projection operations** – Filter and projection operations are now always processed by columns if possible. The engine automatically takes advantage of dictionary encodings where effective.
- **Gathering exchanges** – Improved performance for queries with gathering exchanges.
- **Global aggregations** – Improved performance for some queries that perform filtered global aggregations.
- **GROUPING SETS, CUBE, ROLLUP** – Improved performance for queries involving GROUPING SETS, CUBE or ROLLUP, which you can use to aggregate multiple sets of columns in a single query.
- **Highly selective filters** – Improved the performance of queries with highly selective filters.
- **JOIN and AGGREGATE operations** – The performance of JOIN and AGGREGATE operations has been enhanced.
- **LIKE** – Improved the performance of queries that use LIKE predicates on the columns of information_schema tables.
- **ORDER BY and LIMIT** – Improved plans, performance, and memory usage for queries involving ORDER BY and LIMIT to avoid unnecessary data exchanges.
- **ORDER BY** – ORDER BY operations are now distributed by default, enabling larger ORDER BY clauses to be used.
- **ROW type conversions** – Improved performance when converting between ROW types.
- **Structural types** – Improved performance of queries that process structural types and contain scan, joins, aggregations, or table writes.
- **Table scans** – An optimization rule has been introduced to avoid duplicate table scans in certain cases.
• **UNION** – Improved performance for UNION queries.

**Query planning performance**

• **Planning performance** – Improved planning performance for queries that join multiple tables with a large number of columns.

• **Predicate evaluations** – Improved predicate evaluation performance during predicate pushdown in planning.

• **Predicate pushdown support for casting** – Support predicate pushdown for the `<column> IN <values list>` predicate where values in the values list require casting to match the type of column.

• **Predicate inference and pushdown** – Predicate inference and pushdown extended for queries that use a `<symbol> IN <subquery>` predicate.

• **Timeouts** – Fixed a bug that could in rare cases cause query planning timeouts.

**Join performance**

• **Joins with map columns** – Improved the performance of joins and aggregations that include map columns.

• **Joins with solely non-equality conditions** – Improved the performance of joins with only non-equality conditions by using a nested loop join instead of a hash join.

• **Outer joins** – The join distribution type is now automatically selected for queries involving outer joins.

• **Range over a function joins** – Improved performance of joins where the condition is a range over a function (for example, a JOIN b ON b.x < f(a.x) AND b.x > g(a.x)).

• **Spill-to-disk** – Fixed spill-to-disk related bugs and memory issues to enhance performance and reduce memory errors in JOIN operations.

**Subquery performance**

• **Correlated EXISTS subqueries** – Improved performance of correlated EXISTS subqueries.

• **Correlated subqueries with equalities** – Improved support for correlated subqueries containing equality predicates.

• **Correlated subqueries with inequalities** – Improved performance for correlated subqueries that contain inequalities.

• **Count(*) aggregations over subqueries** – Improved performance of count(*) aggregations over subqueries with known constant cardinality.

• **Outer query filter propagation** – Improved performance of correlated subqueries when filters from the outer query can be propagated to the subquery.

**Function performance**

• **Aggregate window functions** – Improved performance of aggregate window functions.

• **element_at()** – Improved performance of `element_at()` for maps to be constant time rather than proportional to the size of the map.

• **Grouping()** – Improved performance for queries involving `grouping()`.

• **JSON casting** – Improved the performance of casting from JSON to ARRAY or MAP types.

• **Map-returning functions** – Improved performance of functions that return maps.

• **Map-to-map casting** – Improved the performance of map-to-map cast.

• **Min() and max()** – The `min()` and `max()` functions have been optimized to avoid unnecessary object creation, thus reducing garbage collection overhead.
• **row_number()** – Improved performance and memory usage for queries using `row_number()` followed by a filter on the row numbers generated.

• **Window functions** – Improved performance of queries containing window functions with identical `PARTITION BY` and `ORDER BY` clauses.

• **Window functions** – Improved performance of certain window functions (for example, `LAG`) that have similar specifications.

**Geospatial performance**

• **Geometry serialization** – Improved the serialization performance of geometry values.

• **Geospatial functions** – Improved the performance of `ST_Intersects()`, `ST_Contains()`, `ST_Touches()`, `ST_Overlaps()`, `ST_Disjoint()`, `transform_values()`, `ST_XMin()`, `ST_XMax()`, `ST_YMin()`, `ST_YMax()`, `ST_Crosses()`, and `array_intersect()`.

• **ST_Distance()** – Improved performance of join queries involving the `ST_Distance()` function.

• **ST_Intersection()** – Optimized the `ST_Intersection()` function for rectangles aligned with coordinate axes (for example, polygons produced by the `ST_Envelope()` and `bing_tile_polygon()` functions).

**JSON-related improvements**

**Map Functions**

• Improved performance of map subscript from $O(n)$ to $O(1)$ in all cases. Previously, only maps produced by certain functions and readers took advantage of this improvement.

• Added the `map_from_entries()` and `map_entries()` functions.

**Casting**

• Added ability to cast to JSON from REAL, TINYINT or SMALLINT.

• You can now cast JSON to ROW even if the JSON does not contain every field in the ROW.

• Improved performance of `CAST(json_parse(...) AS ...)`.

• Improved the performance of casting from JSON to ARRAY or MAP types.

**New JSON Functions**

• `is_json_scalar()`

**Breaking changes**

Breaking changes include bug fixes, changes to geospatial functions, replaced functions, and the introduction of limits. Improvements in ANSI SQL compliance may break queries that depended on non-standard behavior.

**Bug fixes**

The following changes correct behavioral issues that caused queries to run successfully, but with inaccurate results.

• **fixed_len_byte_array parquet columns are now accepted as DECIMAL** – Queries on Parquet columns of type `fixed_len_byte_array` succeed and return correct values if they are annotated as DECIMAL in the Parquet Schema. Queries on `fixed_len_byte_array` columns without the DECIMAL
annotation fail with an error. Previously, queries on fixed_len_byte_array columns without the DECIMAL annotation succeeded but returned incomprehensible values.

- **json_parse() no longer ignores trailing characters** – Previously, inputs such as \[1,2\]abc would successfully parse as \[1, 2\]. Using trailing characters now produces the error message Cannot convert ‘\[1, 2\]abc’ to JSON.

- **Round() decimal precision corrected** – \(\text{round}(x, d)\) now correctly rounds \(x\) when \(x\) is a DECIMAL or when \(x\) is a DECIMAL with scale 0 and \(d\) is a negative integer. Previously, no rounding occurred in these cases.

- **\(\text{round}(x, d)\) and \(\text{truncate}(x, d)\)** – The parameter \(d\) in the signature of functions \(\text{round}(x, d)\) and \(\text{truncate}(x, d)\) is now of type INTEGER. Previously, \(d\) could be of type BIGINT.

- **Map() with duplicate keys** – \(\text{map()}\) now raises an error on duplicate keys rather than silently producing a corrupted map. Queries that currently construct map values using duplicate keys now fail with an error.

- **map_from_entries() raises an error with null entries** – \(\text{map_from_entries()}\) now raises an error when the input array contains a null entry. Queries that construct a map by passing NULL as a value now fail.

- **Tables** – Tables that have unsupported partition types can no longer be created.

- **Improved numerical stability in statistical functions** – The numerical stability for the statistical functions \(\text{corr()}, \text{covar_samp()}, \text{regr_intercept}(),\) and \(\text{regr_slope}()\) has been improved.

- **TIMESTAMP precision defined in parquet is now respected** – The precision of TIMESTAMP values and the precision defined for the TIMESTAMP column in the Parquet schema must now match. Non-matching precisions result in incorrect timestamps.

- **Time zone information** – Time zone information is now calculated using the java.time package of the Java 1.8 SDK.

- **SUM of INTERVAL_DAY_TO_SECOND and INTERVAL_YEAR_TO_MONTH datatypes** – You can no longer use \(\text{SUM}(\text{NULL})\) directly. In order to use \(\text{SUM}(\text{NULL})\), cast NULL to a data type like BIGINT, DECIMAL, REAL, DOUBLE, INTERVAL_DAY_TO_SECOND or INTERVAL_YEAR_TO_MONTH.

### Changes to geospatial functions

Changes made to geospatial functions include the following.

- **Function name changes** – Some function names have changed. For more information, see Geospatial function name changes in Athena engine version 2 (p. 416).

- **VARBINARY input** – The VARBINARY type is no longer directly supported for input to geospatial functions. For example, to calculate the area of a geometry directly, the geometry must now be input in either VARCHAR or GEOMETRY format. The workaround is to use transform functions, as in the following examples.

  - To use \(\text{ST_area()}\) to calculate the area for VARBINARY input in Well-Known Binary (WKB) format, pass the input to \(\text{ST_GeomFromBinary()}\) first, for example:

    ```sql
    \text{ST_area(\text{ST_GeomFromBinary(<wkb_varbinary_value>))})}
    ```

  - To use \(\text{ST_area()}\) to calculate the area for VARBINARY input in legacy binary format, pass the same input to the \(\text{ST_GeomFromLegacyBinary()}\) function first, for example:

    ```sql
    \text{ST_area(\text{ST_GeomFromLegacyBinary(<legacy_varbinary_value>))})}
    ```

- **\(\text{ST_ExteriorRing()}\) and \(\text{ST_Polygon()}\)** – \(\text{ST_ExteriorRing()}\) (p. 409) and \(\text{ST_Polygon()}\) (p. 406) now accept only polygons as inputs. Previously, these functions erroneously accepted other geometries.

- **\(\text{ST_Distance()}\)** – As required by the SQL/MM specification, the \(\text{ST_Distance()}\) (p. 411) function now returns NULL if one of the inputs is an empty geometry. Previously, NaN was returned.
ANSI SQL compliance

The following syntax and behavioral issues have been corrected to follow the ANSI SQL standard.

- **Cast() operations** – Cast() operations from REAL or DOUBLE to DECIMAL now conform to the SQL standard. For example, `cast (double '100000000000000000000000000000000' as decimal(38))` previously returned `100000000000000000000000000000000` but now returns `100000000000000000000000000000000`.

- **JOIN ... USING** – JOIN ... USING now conforms to standard SQL semantics. Previously, JOIN ... USING required qualifying the table name in columns, and the column from both tables would be present in the output. Table qualifications are now invalid and the column is present only once in the output.

- **ROW type literals removed** – The ROW type literal format `ROW<int, int>(1, 2)` is no longer supported. Use the syntax `ROW(1 int, 2 int)` instead.

- **Log() function** – Previously, in violation of the SQL standard, the order of the arguments in the log() function was reversed. This caused log() to return incorrect results when queries were translated to or from other SQL implementations. The equivalent to `log(x, b)` is now correctly `ln(x) / ln(b)`.

- **Grouped aggregation semantics** – Grouped aggregations use `IS NOT DISTINCT FROM` semantics rather than equality semantics. Grouped aggregations now return correct results and show improved performance when grouping on NaN floating point values. Grouping on map, list, and row types that contain nulls is supported.

- **Types with quotation marks are no longer allowed** – In accordance with the ANSI SQL standard, data types can no longer be enclosed in quotation marks. For example, `SELECT "date" '2020-02-02'` is no longer a valid query. Instead, use the syntax `SELECT date '2020-02-02`.

- **Anonymous row field access** – Anonymous row fields can no longer be accessed by using the syntax `[.field0, .field1, ...]`.

- **Complex grouping operations** – The complex grouping operations GROUPING SETS, CUBE, and ROLLUP do not support grouping on expressions composed of input columns. Only column names are allowed.

Replaced functions

The following functions are no longer supported and have been replaced by syntax that produces the same results.

- **information_schema.__internal_partitions__** – The usage of __internal_partitions__ is no longer supported in Athena engine version 2. For equivalent syntax, use `SELECT * FROM "<table_name>"$partitions"` or SHOW PARTITIONS. For more information, see Listing partitions for a specific table (p. 444).

- **Replaced geospatial functions** – For a list of geospatial functions whose names have changed, see Geospatial function name changes in Athena engine version 2 (p. 416).

Limits

The following limits were introduced in Athena engine version 2 to ensure that queries do not fail due to resource limitations. These limits are not configurable by users.

- **Number of result elements** – The number of result elements n is restricted to 10,000 or less for the following functions: min(col, n), max(col, n), min_by(col1, col2, n), and max_by(col1, col2, n).

- **GROUPING SETS** – The maximum number of slices in a grouping set is 2048.

- **Maximum text file line length** – The default maximum line length for text files is 100 MB.
• **Sequence function maximum result size** – The maximum result size of a sequence function is 50000 entries. For example, `SELECT sequence(0, 45000, 1)` succeeds, but `SELECT sequence(0, 55000, 1)` fails with the error message "The result of the sequence function must not have more than 50000 entries. This limit applies to all input types for sequence functions, including timestamps."
SQL reference for Amazon Athena

Amazon Athena supports a subset of Data Definition Language (DDL) and Data Manipulation Language (DML) statements, functions, operators, and data types. With some exceptions, Athena DDL is based on HiveQL DDL. For information about Athena engine versions, see Athena engine versioning (p. 669).

Topics
- Data types in Amazon Athena (p. 690)
- DML queries, functions, and operators (p. 692)
- DDL statements (p. 730)
- Considerations and limitations for SQL queries in Amazon Athena (p. 764)

Data types in Amazon Athena

When you run CREATE TABLE (p. 740), you specify column names and the data type that each column can contain. Athena supports the data types listed below. For information about the data type mappings that the JDBC driver supports between Athena, JDBC, and Java, see Data types in the JDBC Driver Installation and Configuration Guide. For information about the data type mappings that the ODBC driver supports between Athena and SQL, see Data types in the ODBC Driver Installation and Configuration Guide.

- **boolean** – Values are true and false.
- **tinyint** – A 8-bit signed integer in two's complement format, with a minimum value of \(-2^7\) and a maximum value of \(2^7-1\).
- **smallint** – A 16-bit signed integer in two's complement format, with a minimum value of \(-2^{15}\) and a maximum value of \(2^{15}-1\).
- **int** and **integer** – Athena uses different expressions for integer depending on the type of query. **int** – In Data Definition Language (DDL) queries like CREATE TABLE, use the int data type. **integer** – In DML queries like SELECT * FROM, use the integer data type. integer is represented as a 32-bit signed value in two's complement format, with a minimum value of \(-2^{31}\) and a maximum value of \(2^{31}-1\). To ensure compatibility with business analytics applications, the JDBC driver returns the integer type.
- **bigint** – A 64-bit signed integer in two's complement format, with a minimum value of \(-2^{63}\) and a maximum value of \(2^{63}-1\).
- **double** – A 64-bit signed double-precision floating point number. The range is 4.94065645841246544e-324d to 1.79769313486231570e+308d, positive or negative. double follows the IEEE Standard for Floating-Point Arithmetic (IEEE 754).
- **float** – A 32-bit signed single-precision floating point number. The range is 1.40129846432481707e-45 to 3.40282346638528860e+38, positive or negative. float follows the IEEE Standard for Floating-Point Arithmetic (IEEE 754). Equivalent to the real in Presto. In Athena, use float in DDL statements like CREATE TABLE and real in SQL functions like SELECT CAST. The AWS Glue crawler returns values in float, and Athena translates real and float types internally (see the June 5, 2018 (p. 831) release notes).
- **decimal**(precision, scale) – precision is the total number of digits. scale (optional) is the number of digits in fractional part with a default of 0. For example, use these type definitions: decimal(11,5), decimal(15). The maximum value for precision is 38, and the maximum value for scale is 38.
To specify decimal values as literals, such as when selecting rows with a specific decimal value in a query DDL expression, specify the decimal type definition, and list the decimal value as a literal (in single quotes) in your query, as in this example: `decimal_value = decimal '0.12'`.

- **char** – Fixed length character data, with a specified length between 1 and 255, such as `char(10)`. For more information, see CHAR Hive data type.

  **Note**
  To use the `substr` function to return a substring of specified length from a char data type, you must first cast the char value as a varchar, as in the following example.

  ```
  substr(cast(col1 as varchar), 1, 4)
  ```

- **varchar** – Variable length character data, with a specified length between 1 and 65535, such as `varchar(10)`. For more information, see VARCHAR Hive data type.

- **string** – A string literal enclosed in single or double quotes. For more information, see STRING Hive data type.

  **Note**
  Non-string data types cannot be cast to string in Athena; cast them to varchar instead.

- **binary** – Used for data in Parquet.

- **date** – A date in ISO format, such as `YYYY-MM-DD`. For example, `date '2008-09-15'`. An exception is the OpenCSVSerDe, which uses the number of days elapsed since January 1, 1970. For more information, see OpenCSVSerDe for processing CSV (p. 329).

- **timestamp** – Date and time instant in a `java.sql.Timestamp` compatible format up to a maximum resolution of milliseconds, such as `yyyy-MM-dd HH:mm:ss[.f...]`. For example, `timestamp '2008-09-15 03:04:05.324'`. An exception is the OpenCSVSerDe, which uses timestamp data in the UNIX numeric format (for example, `1579059880000`). For more information, see OpenCSVSerDe for processing CSV (p. 329).

- **array<data_type>** – An array of the given component type.

  Example

  ```
  CREATE TABLE table array_table (c1 array<integer>) LOCATION '...';
  INSERT INTO array_table values(ARRAY[1,2,3]);
  ```

- **map<primitive_type, data_type>** – A map between the given component types.

  Example

  ```
  CREATE TABLE map_table(c1 map<string, integer>) LOCATION '...';
  INSERT INTO map_table values(MAP(ARRAY['foo', 'bar'], ARRAY[1, 2]));
  ```

- **struct<col_name : data_type [comment col_comment] , ...>** – A collection of elements of different component types.

  Example

  ```
  CREATE TABLE struct_table(c1 struct<name:varchar(10), age:integer>) LOCATION '...';
  INSERT INTO struct_table SELECT CAST(ROW('Bob', 38) AS ROW(name VARCHAR(10), age INTEGER));
  ```
DML queries, functions, and operators

The Athena DML query engine generally supports Trino and Presto syntax and adds its own improvements. Athena does not support all Trino or Presto features. For more information, see the topics for specific statements in this section and Considerations and limitations (p. 764). For information about functions, see Functions in Amazon Athena (p. 719). For information about Athena engine versions, see Athena engine versioning (p. 669).

For information about DDL statements, see DDL statements (p. 730). For a list of unsupported DDL statements, see Unsupported DDL (p. 731).

SELECT

Retrieves rows of data from zero or more tables.

Note
This topic provides summary information for reference. Comprehensive information about using SELECT and the SQL language is beyond the scope of this documentation. For information about using SQL that is specific to Athena, see Considerations and limitations for SQL queries in Amazon Athena (p. 764) and Running SQL queries using Amazon Athena (p. 339). For an example of creating a database, creating a table, and running a SELECT query on the table in Athena, see Getting started (p. 9).

Synopsis

```
[ WITH with_query [, ....] ]
SELECT [ ALL | DISTINCT ] select_expression [, ...]
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY [ ALL | DISTINCT ] grouping_element [, ...] ]
[ HAVING condition ]
[ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
[ ORDER BY expression [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ] [, ... ]]
[ OFFSET count [ ROW | ROWS ]]
[ LIMIT [ count | ALL ]]
```

Note
Reserved words in SQL SELECT statements must be enclosed in double quotes. For more information, see List of reserved keywords in SQL SELECT statements (p. 256).

Parameters

```
[ WITH with_query [, ....] ]
```

You can use WITH to flatten nested queries, or to simplify subqueries.

Using the WITH clause to create recursive queries is not supported.

The WITH clause precedes the SELECT list in a query and defines one or more subqueries for use within the SELECT query.

Each subquery defines a temporary table, similar to a view definition, which you can reference in the FROM clause. The tables are used only when the query runs.

with_query syntax is:
SELECT subquery_table_name [ ( column_name [, ...] ) ] AS (subquery)

Where:
- subquery_table_name is a unique name for a temporary table that defines the results of the WITH clause subquery. Each subquery must have a table name that can be referenced in the FROM clause.
- column_name [, ...] is an optional list of output column names. The number of column names must be equal to or less than the number of columns defined by subquery.
- subquery is any query statement.

[ ALL | DISTINCT ] select_expr

determines the rows to be selected.

ALL is the default. Using ALL is treated the same as if it were omitted; all rows for all columns are selected and duplicates are kept.

Use DISTINCT to return only distinct values when a column contains duplicate values.

FROM from_item [ ...]

Indicates the input to the query, where from_item can be a view, a join construct, or a subquery as described below.

The from_item can be either:
- table_name [ [ AS ] alias [ (column_alias [, ...]) ] ]

  Where table_name is the name of the target table from which to select rows, alias is the name to give the output of the SELECT statement, and column_alias defines the columns for the alias specified.

- OR-
- join_type from_item [ ON join_condition | USING ( join_column [, ...] ) ]

  Where join_type is one of:
  - [ INNER ] JOIN
  - LEFT [ OUTER ] JOIN
  - RIGHT [ OUTER ] JOIN
  - FULL [ OUTER ] JOIN
  - CROSS JOIN

  ON join_condition | USING ( join_column [, ...]) Where using join_condition allows you to specify column names for join keys in multiple tables, and using join_column requires join_column to exist in both tables.

[ WHERE condition ]

Filters results according to the condition you specify, where condition generally has the following syntax.

column_name operator value [[AND | OR] column_name operator value] ...]

The operator can be one of the comparators =, >, >=, <, <=, <>, !=.

The following subquery expressions can also be used in the WHERE clause.
- [NOT] BETWEEN integer_A AND integer_B – Specifies a range between two integers, as in the following example. If the column datatype is varchar, the column must be cast to integer first.
SELECT DISTINCT processid FROM "webdata"."impressions"
WHERE cast(processid as int) BETWEEN 1500 and 1800
ORDER BY processid

• [NOT] LIKE value – Searches for the pattern specified. Use the percent sign (%) as a wildcard character, as in the following example.

    SELECT * FROM "webdata"."impressions"
    WHERE referrer LIKE '%.org'

• [NOT] IN (value[, value[, ...]]) – Specifies a list of possible values for a column, as in the following example.

    SELECT * FROM "webdata"."impressions"
    WHERE referrer IN ('example.com','example.net','example.org')

[ GROUP BY [ ALL | DISTINCT ] grouping_expressions [ , ... ] ]

Divides the output of the SELECT statement into rows with matching values.

ALL and DISTINCT determine whether duplicate grouping sets each produce distinct output rows. If omitted, ALL is assumed.

grouping_expressions allow you to perform complex grouping operations. You can use complex grouping operations to perform analysis that requires aggregation on multiple sets of columns in a single query.

The grouping_expressions element can be any function, such as SUM, AVG, or COUNT, performed on input columns.

GROUP BY expressions can group output by input column names that don't appear in the output of the SELECT statement.

All output expressions must be either aggregate functions or columns present in the GROUP BY clause.

You can use a single query to perform analysis that requires aggregating multiple column sets.

Athena supports complex aggregations using GROUPING SETS, CUBE and ROLLUP. GROUP BY GROUPING SETS specifies multiple lists of columns to group on. GROUP BY CUBE generates all possible grouping sets for a given set of columns. GROUP BY ROLLUP generates all possible subtotals for a given set of columns. Complex grouping operations do not support grouping on expressions composed of input columns. Only column names are allowed.

You can often use UNION ALL to achieve the same results as these GROUP BY operations, but queries that use GROUP BY have the advantage of reading the data one time, whereas UNION ALL reads the underlying data three times and may produce inconsistent results when the data source is subject to change.

[ HAVING condition ]

Used with aggregate functions and the GROUP BY clause. Controls which groups are selected, eliminating groups that don't satisfy condition. This filtering occurs after groups and aggregates are computed.

[[ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] union_query ]]

UNION, INTERSECT, and EXCEPT combine the results of more than one SELECT statement into a single query. ALL or DISTINCT control the uniqueness of the rows included in the final result set.
UNION combines the rows resulting from the first query with the rows resulting from the second query. To eliminate duplicates, UNION builds a hash table, which consumes memory. For better performance, consider using UNION ALL if your query does not require the elimination of duplicates. Multiple UNION clauses are processed left to right unless you use parentheses to explicitly define the order of processing.

INTERSECT returns only the rows that are present in the results of both the first and the second queries.

EXCEPT returns the rows from the results of the first query, excluding the rows found by the second query.

ALL causes all rows to be included, even if the rows are identical.

DISTINCT causes only unique rows to be included in the combined result set.

[ ORDER BY expression [ ASC | DESC ] [ NULLS FIRST | NULLS LAST] [, ...] ]

Sorts a result set by one or more output expression.

When the clause contains multiple expressions, the result set is sorted according to the first expression. Then the second expression is applied to rows that have matching values from the first expression, and so on.

Each expression may specify output columns from SELECT or an ordinal number for an output column by position, starting at one.

ORDER BY is evaluated as the last step after any GROUP BY or HAVING clause. ASC and DESC determine whether results are sorted in ascending or descending order.

The default null ordering is NULLS LAST, regardless of ascending or descending sort order.

[ OFFSET count [ ROW | ROWS ] ]

Use the OFFSET clause to discard a number of leading rows from the result set. If the ORDER BY clause is present, the OFFSET clause is evaluated over a sorted result set, and the set remains sorted after the skipped rows are discarded. If the query has no ORDER BY clause, it is arbitrary which rows are discarded. If the count specified by OFFSET equals or exceeds the size of the result set, the final result is empty.

LIMIT [ count | ALL ]

Restricts the number of rows in the result set to count. LIMIT ALL is the same as omitting the LIMIT clause. If the query has no ORDER BY clause, the results are arbitrary.

TABLESAMPLE [ BERNOULLI | SYSTEM ] (percentage)

Optional operator to select rows from a table based on a sampling method.

BERNOULLI selects each row to be in the table sample with a probability of percentage. All physical blocks of the table are scanned, and certain rows are skipped based on a comparison between the sample percentage and a random value calculated at runtime.

With SYSTEM, the table is divided into logical segments of data, and the table is sampled at this granularity.

Either all rows from a particular segment are selected, or the segment is skipped based on a comparison between the sample percentage and a random value calculated at runtime. SYSTEM sampling is dependent on the connector. This method does not guarantee independent sampling probabilities.
[UNNEST (array_or_map) [WITH ORDINALITY]]

Expands an array or map into a relation. Arrays are expanded into a single column. Maps are expanded into two columns (key, value).

You can use UNNEST with multiple arguments, which are expanded into multiple columns with as many rows as the highest cardinality argument.

Other columns are padded with nulls.

The WITH ORDINALITY clause adds an ordinality column to the end.

UNNEST is usually used with a JOIN and can reference columns from relations on the left side of the JOIN.

Getting the file locations for source data in Amazon S3

To see the Amazon S3 file location for the data in a table row, you can use "$path" in a SELECT query, as in the following example:

```
SELECT "$path" FROM "my_database"."my_table" WHERE year=2019;
```

This returns a result like the following:

```
s3://awsexamplebucket/datasets_mytable/year=2019/data_file1.json
```

To return a sorted, unique list of the S3 filename paths for the data in a table, you can use SELECT DISTINCT and ORDER BY, as in the following example.

```
SELECT DISTINCT "$path" AS data_source_file
FROM sampledb.elb_logs
ORDER By data_source_file ASC
```

To return only the filenames without the path, you can pass "$path" as a parameter to an regexp_extract function, as in the following example.

```
SELECT DISTINCT regexp_extract("$path", '[^/]\+$') AS data_source_file
FROM sampledb.elb_logs
ORDER By data_source_file ASC
```

To return the data from a specific file, specify the file in the WHERE clause, as in the following example.

```
SELECT *,"$path" FROM my_database.my_table WHERE "$path" = 's3://awsexamplebucket/my_table/my_partition/file-01.csv'
```

For more information and examples, see the Knowledge Center article How can I see the Amazon S3 source file for a row in an Athena table?

Escaping single quotes

To escape a single quote, precede it with another single quote, as in the following example. Do not confuse this with a double quote.

```
Select 'O''Reilly'
```
Results

Additional resources

For more information about using SELECT statements in Athena, see the following resources.

<table>
<thead>
<tr>
<th>For information about this</th>
<th>See this</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running queries in Athena</td>
<td>Running SQL queries using Amazon Athena (p. 339)</td>
</tr>
<tr>
<td>Using SELECT to create a table</td>
<td>Creating a table from query results (CTAS) (p. 281)</td>
</tr>
<tr>
<td>Inserting data from a SELECT query into another table</td>
<td>INSERT INTO (p. 697)</td>
</tr>
<tr>
<td>Using built-in functions in SELECT statements</td>
<td>Functions in Amazon Athena (p. 719)</td>
</tr>
<tr>
<td>Using user defined functions in SELECT statements</td>
<td>Querying with user defined functions (p. 434)</td>
</tr>
<tr>
<td>Querying Data Catalog metadata</td>
<td>Querying AWS Glue Data Catalog (p. 442)</td>
</tr>
</tbody>
</table>

**INSERT INTO**

Inserts new rows into a destination table based on a SELECT query statement that runs on a source table, or based on a set of VALUES provided as part of the statement. When the source table is based on underlying data in one format, such as CSV or JSON, and the destination table is based on another format, such as Parquet or ORC, you can use INSERT INTO queries to transform selected data into the destination table's format.

**Considerations and limitations**

Consider the following when using INSERT queries with Athena.

- When running an INSERT query on a table with underlying data that is encrypted in Amazon S3, the output files that the INSERT query writes are not encrypted by default. We recommend that you encrypt INSERT query results if you are inserting into tables with encrypted data.

  For more information about encrypting query results using the console, see Encrypting Athena query results stored in Amazon S3 (p. 523). To enable encryption using the AWS CLI or Athena API, use the EncryptionConfiguration properties of the StartQueryExecution action to specify Amazon S3 encryption options according to your requirements.

- For INSERT INTO statements, the expected bucket owner setting does not apply to the destination table location in Amazon S3. The expected bucket owner setting applies only to the Amazon S3 output location that you specify for Athena query results. For more information, see Specifying a query result location using the Athena console (p. 344).

- For ACID compliant INSERT INTO statements, see the INSERT INTO section of Updating Iceberg table data (p. 513).

**Supported formats and SerDes**

You can run an INSERT query on tables created from data with the following formats and SerDes.
### Data format and SerDe

<table>
<thead>
<tr>
<th>Data format</th>
<th>SerDe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avro</td>
<td>org.apache.hadoop.hive.serde2.avro.AvroSerDe</td>
</tr>
<tr>
<td>JSON</td>
<td>org.apache.hive.hcatalog.data.JsonSerDe</td>
</tr>
<tr>
<td>ORC</td>
<td>org.apache.hadoop.hive.ql.io.orc.OrcSerde</td>
</tr>
<tr>
<td>Parquet</td>
<td>org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe</td>
</tr>
<tr>
<td>Text file</td>
<td>org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe</td>
</tr>
</tbody>
</table>

**Note**

CSV, TSV, and custom-delimited files are supported.

### Bucketed tables not supported

`INSERT INTO` is not supported on bucketed tables. For more information, see Bucketing vs partitioning (p. 284).

### Federated queries not supported

`INSERT INTO` is not supported for federated queries. Attempting to do so may result in the error message This operation is currently not supported for external catalogs. For information about federated queries, see Using Amazon Athena Federated Query (p. 60).

### Partitioning

Consider the points in this section when using partitioning with `INSERT INTO` or `CREATE TABLE AS SELECT` queries.

**Limits**

The `INSERT INTO` statement supports writing a maximum of 100 partitions to the destination table. If you run the `SELECT` clause on a table with more than 100 partitions, the query fails unless the `SELECT` query is limited to 100 partitions or fewer.

For information about working around this limitation, see Using CTAS and `INSERT INTO` to create a table with more than 100 partitions (p. 295).

**Column ordering**

`INSERT INTO` or `CREATE TABLE AS SELECT` statements expect the partitioned column to be the last column in the list of projected columns in a `SELECT` statement.

If the source table is non-partitioned, or partitioned on different columns compared to the destination table, queries like `INSERT INTO destination_table SELECT * FROM source_table` consider the values in the last column of the source table to be values for a partition column in the destination table. Keep this in mind when trying to create a partitioned table from a non-partitioned table.

### Resources

For more information about using `INSERT INTO` with partitioning, see the following resources:

- For inserting partitioned data into a partitioned table, see Using CTAS and `INSERT INTO` to create a table with more than 100 partitions (p. 295).
- For inserting unpartitioned data into a partitioned table, see Using CTAS and `INSERT INTO` for ETL and data analysis (p. 290).
Files written to Amazon S3

Athena writes files to source data locations in Amazon S3 as a result of the INSERT command. Each INSERT operation creates a new file, rather than appending to an existing file. The file locations depend on the structure of the table and the SELECT query, if present. Athena generates a data manifest file for each INSERT query. The manifest tracks the files that the query wrote. It is saved to the Athena query result location in Amazon S3. For more information, see Identifying query output files (p. 352).

Locating orphaned files

If a CTAS or INSERT INTO statement fails, it is possible that orphaned data are left in the data location. Because Athena does not delete any data (even partial data) from your bucket, you might be able to read this partial data in subsequent queries. To locate orphaned files for inspection or deletion, you can use the data manifest file that Athena provides to track the list of files to be written. For more information, see Identifying query output files (p. 352) and DataManifestLocation.

INSERT INTO...SELECT

Specifies the query to run on one table, source_table, which determines rows to insert into a second table, destination_table. If the SELECT query specifies columns in the source_table, the columns must precisely match those in the destination_table.

For more information about SELECT queries, see SELECT (p. 692).

Synopsis

```
INSERT INTO destination_table
SELECT select_query
FROM source_table_or_view
```

Examples

Select all rows in the vancouver_pageviews table and insert them into the canada_pageviews table:

```
INSERT INTO canada_pageviews
SELECT *
FROM vancouver_pageviews;
```

Select only those rows in the vancouver_pageviews table where the date column has a value between 2019-07-01 and 2019-07-31, and then insert them into canada_july_pageviews:

```
INSERT INTO canada_july_pageviews
SELECT *
FROM vancouver_pageviews
WHERE date BETWEEN '2019-07-01'
AND '2019-07-31';
```

Select the values in the city and state columns in the cities_world table only from those rows with a value of usa in the country column and insert them into the city and state columns in the cities_usa table:

```
INSERT INTO cities_usa (city,state)
SELECT city,state
FROM cities_world
WHERE country='usa'
```
**INSERT INTO...VALUES**

Inserts rows into an existing table by specifying columns and values. Specified columns and associated data types must precisely match the columns and data types in the destination table.

**Important**

We do not recommend inserting rows using VALUES because Athena generates files for each INSERT operation. This can cause many small files to be created and degrade the table's query performance. To identify files that an INSERT query creates, examine the data manifest file. For more information, see Working with query results, recent queries, and output files (p. 344).

**Synopsis**

```
INSERT INTO destination_table [(col1,col2,...)]
VALUES (col1value,col2value,...)[,
(col1value,col2value,...)][,
...
```

**Examples**

In the following examples, the cities table has three columns: id, city, state, state_motto. The id column is type INT and all other columns are type VARCHAR.

Insert a single row into the cities table, with all column values specified:

```
INSERT INTO cities
VALUES (1,'Lansing','MI','Si quaeris peninsulam amoenam circumspice')
```

Insert two rows into the cities table:

```
INSERT INTO cities
VALUES (1,'Lansing','MI','Si quaeris peninsulam amoenam circumspice'),
(3,'Boise','ID','Esto perpetua')
```

**DELETE**

Deletes rows in an Apache Iceberg table. DELETE is transactional and is supported only for Apache Iceberg tables.

**Synopsis**

To delete the rows from an Iceberg table, use the following syntax.

```
DELETE FROM [db_name.]table_name [WHERE predicate]
```

For more information and examples, see the DELETE section of Updating Iceberg table data (p. 513).

**UPDATE**

Updates rows in an Apache Iceberg table. UPDATE is transactional and is supported only for Apache Iceberg tables.

**Synopsis**

To update the rows in an Iceberg table, use the following syntax.
MERGE INTO

Conditionally updates, deletes, or inserts rows into an Apache Iceberg table. A single statement can combine update, delete, and insert actions.

**Note**

MERGE INTO is transactional and is supported only for Apache Iceberg tables in Athena engine version 3.

**Synopsis**

To conditionally update, delete, or insert rows from an Iceberg table, use the following syntax.

```
MERGE INTO target_table [ [ AS ] target_alias ]
USING { source_table | query } [ [ AS ] source_alias ]
ON search_condition
when_clause [...]
```

The *when_clause* is one of the following:

- WHEN MATCHED [ AND condition ]
  THEN DELETE

- WHEN MATCHED [ AND condition ]
  THEN UPDATE SET ( column = expression [, ...] )

- WHEN NOT MATCHED [ AND condition ]
  THEN INSERT [ column_list ] VALUES (expression, ...)

MERGE supports an arbitrary number of WHEN clauses with different MATCHED conditions. The condition clauses execute the DELETE, UPDATE or INSERT operation in the first WHEN clause selected by the MATCHED state and the match condition.

For each source row, the WHEN clauses are processed in order. Only the first matching WHEN clause is executed. Subsequent clauses are ignored. A user error is raised when a single target table row matches more than one source row.

If a source row is not matched by any WHEN clause and there is no WHEN NOT MATCHED clause, the source row is ignored.

In WHEN clauses that have UPDATE operations, the column value expressions can refer to any field of the target or the source. In the NOT MATCHED case, the INSERT expressions can refer to any field of the source.

For examples, see [Updating Iceberg table data (p. 513)](#).

**OPTIMIZE**

Optimizes rows in an Apache Iceberg table by rewriting data files into a more optimized layout based on their size and number of associated delete files.
Note
OPTIMIZE is transactional and is supported only for Apache Iceberg tables.

Syntax
The following syntax summary shows how to optimize data layout for an Iceberg table.

```
OPTIMIZE [db_name.]table_name REWRITE DATA USING BIN_PACK
[WHERE predicate]
```

The compaction action is charged by the amount of data scanned during the rewrite process. The REWRITE DATA action uses predicates to select for files that contain matching rows. If any row in the file matches the predicate, the file is selected for optimization. Thus, to control the number of files affected by the compaction operation, you can specify a WHERE clause.

Configuring compaction properties
To control the size of the files to be selected for compaction and the resulting file size after compaction, you can use table property parameters. You can use the ALTER TABLE SET PROPERTIES (p. 508) command to configure the related table properties (p. 505).

For more information and examples, see the OPTIMIZE section of Updating Iceberg table data (p. 513).

VACUUM
The VACUUM statement performs table maintenance on Apache Iceberg tables by removing no longer needed data files.

Note
VACUUM is transactional and is supported only for Apache Iceberg tables in Athena engine version 3.

Synopsis
To remove data files no longer needed for an Iceberg table, use the following syntax.

```
VACUUM target_table
```

Running the VACUUM statement on Iceberg tables is recommended to remove data files that are no longer relevant and to reduce metadata size and storage consumption.

Warning
If you run a snapshot expiration operation, you can no longer time travel to expired snapshots.

VACUUM performs the following operations:

- Removes snapshots that are older than the amount of time that is specified by the vacuum_max_snapshot_age_seconds table property. By default, this property is set to 432000 seconds (5 days).
- Removes snapshots that are not within the period to be retained that are in excess of the number specified by the vacuum_min_snapshots_to_keep table property. The default is 1.

You can specify these table properties in your CREATE TABLE statement. After the table has been created, you can use the ALTER TABLE SET PROPERTIES (p. 508) statement to update them.

- Removes any metadata and data files that are unreachable as a result of the snapshot removal.
• Removes orphan files that are older than the time specified in the vacuum_max_snapshot_age_seconds table property. Orphan files are files in the table’s data directory that are not part of the table state.

For more information about creating and managing Apache Iceberg tables in Athena, see Creating Iceberg tables (p. 504) and Managing Iceberg tables (p. 507).

Using EXPLAIN and EXPLAIN ANALYZE in Athena

The EXPLAIN statement shows the logical or distributed execution plan of a specified SQL statement, or validates the SQL statement. You can output the results in text format or in a data format for rendering into a graph.

Note
You can view graphical representations of logical and distributed plans for your queries in the Athena console without using the EXPLAIN syntax. For more information, see Viewing execution plans for SQL queries (p. 340).

The EXPLAIN ANALYZE statement shows both the distributed execution plan of a specified SQL statement and the computational cost of each operation in a SQL query. You can output the results in text or JSON format.

Considerations and limitations

The EXPLAIN and EXPLAIN ANALYZE statements in Athena have the following limitations.

• Because EXPLAIN queries do not scan any data, Athena does not charge for them. However, because EXPLAIN queries make calls to AWS Glue to retrieve table metadata, you may incur charges from Glue if the calls go above the free tier limit for glue.
• Because EXPLAIN ANALYZE queries are executed, they do scan data, and Athena charges for the amount of data scanned.
• Row or cell filtering information defined in Lake Formation and query stats information are not shown in the output of EXPLAIN and EXPLAIN ANALYZE.

EXPLAIN syntax

EXPLAIN [ ( option [, ...]) ] statement

option can be one of the following:

FORMAT { TEXT | GRAPHVIZ | JSON }
TYPE { LOGICAL | DISTRIBUTED | VALIDATE | IO }

If the FORMAT option is not specified, the output defaults to TEXT format. The IO type provides information about the tables and schemas that the query reads. IO is supported only in Athena engine version 2 and can be returned only in JSON format.

EXPLAIN ANALYZE syntax

In addition to the output included in EXPLAIN, EXPLAIN ANALYZE output also includes runtime statistics for the specified query such as CPU usage, the number of rows input, and the number of rows output.
EXPLAIN ANALYZE [ ( option [, ... ] ) ] statement

option can be one of the following:

FORMAT { TEXT | JSON }

If the FORMAT option is not specified, the output defaults to TEXT format. Because all queries for EXPLAIN ANALYZE are DISTRIBUTED, the TYPE option is not available for EXPLAIN ANALYZE.

statement can be one of the following:

SELECT
CREATE TABLE AS SELECT
INSERT
UNLOAD

EXPLAIN examples

The following examples for EXPLAIN progress from the more straightforward to the more complex.

EXPLAIN example 1: Use the EXPLAIN statement to show a query plan in text format

In the following example, EXPLAIN shows the execution plan for a SELECT query on Elastic Load Balancing logs. The format defaults to text output.

EXPLAIN
SELECT
  request_timestamp,
  elb_name,
  request_ip
FROM sampledb.elb_logs;

Results

- Output[request_timestamp, elb_name, request_ip] => [[request_timestamp, elb_name, request_ip]]
- RemoteExchange[GATHER] => [[request_timestamp, elb_name, request_ip]]
- TableScan[awsdatacatalog:HiveTableHandle{schemaName=sampledb, tableName=elb_logs, analyzePartitionValues=Optional.empty}] => [[request_timestamp, elb_name, request_ip]]
  LAYOUT: sampledb.elb_logs
  request_ip := request_ip:string:2:REGULAR
  request_timestamp := request_timestamp:string:0:REGULAR
  elb_name := elb_name:string:1:REGULAR

EXPLAIN example 2: Graph a query plan

You can use the Athena console to graph a query plan for you. Enter a SELECT statement like the following into the Athena query editor, and then choose EXPLAIN.

SELECT
  c.c_custkey,
  o.o_orderkey,
The `EXPLAIN` page of the Athena query editor opens and shows you a distributed plan and a logical plan for the query. The following graph shows the logical plan for the example.

`EXPLAIN` or `EXPLAIN ANALYZE` on your query and view the results.

**Important**
Currently, some partition filters may not be visible in the nested operator tree graph even though Athena does apply them to your query. To verify the effect of such filters, run `EXPLAIN` or `EXPLAIN ANALYZE` on your query and view the results.
EXPLAIN example 3: Use the EXPLAIN statement to verify partition pruning

When you use a filtering predicate on a partitioned key to query a partitioned table, the query engine applies the predicate to the partitioned key to reduce the amount of data read.

The following example uses an EXPLAIN query to verify partition pruning for a SELECT query on a partitioned table. First, a CREATE TABLE statement creates the tpch100.orders_partitioned table. The table is partitioned on column o_orderdate.

```
CREATE TABLE `tpch100.orders_partitioned`(
  `o_orderkey` int,
  `o_custkey` int,
  `o_orderstatus` string,
  `o_totalprice` double,
  `o_orderpriority` string,
  `o_clerk` string,
  `o_shippriority` int,
  `o_comment` string)
PARTITIONED BY (o_orderdate string)
STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat'
LOCATION 's3://<your_s3_bucket>/<your_directory_path>/'
```

The tpch100.orders_partitioned table has several partitions on o_orderdate, as shown by the SHOW PARTITIONS command.

```
SHOW PARTITIONS tpch100.orders_partitioned;
```

```
o_orderdate=1994
o_orderdate=1995
o_orderdate=1996
o_orderdate=1997
o_orderdate=1998
o_orderdate=2015
```

The following EXPLAIN query verifies partition pruning on the specified SELECT statement.

```
EXPLAIN
SELECT
  o_orderkey,
  o_custkey,
  o_orderdate
FROM tpch100.orders_partitioned
WHERE o_orderdate = '1995'
```

Results

```
Query Plan
- Output[o_orderkey, o_custkey, o_orderdate] => [[o_orderkey, o_custkey, o_orderdate]]
```
The bold text in the result shows that the predicate \( o\_orderdate = '1995' \) was applied on the PARTITION_KEY.

**EXPLAIN example 4: Use an EXPLAIN query to check the join order and join type**

The following EXPLAIN query checks the SELECT statement's join order and join type. Use a query like this to examine query memory usage so that you can reduce the chances of getting an EXCEEDED\_LOCAL\_MEMORY\_LIMIT error.

```sql
EXPLAIN (TYPE DISTRIBUTED)
SELECT
c.c_custkey,
o.o_orderkey,
o.o_orderstatus
FROM tpch100.customer c
JOIN tpch100.orders o
ON c.c_custkey = o.o_custkey
WHERE c.c_custkey = 123
```

### Results

<table>
<thead>
<tr>
<th>Query Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fragment 0 [SINGLE]</strong></td>
</tr>
<tr>
<td>Output layout: [c_custkey, o_orderkey, o_orderstatus]</td>
</tr>
<tr>
<td>Output partitioning: SINGLE [ ]</td>
</tr>
<tr>
<td>Stage Execution Strategy: UNGROUPED_EXECUTION</td>
</tr>
<tr>
<td>- Output[c_custkey, o_orderkey, o_orderstatus] =&gt; [[c_custkey, o_orderkey, o_orderstatus]]</td>
</tr>
<tr>
<td>- RemoteSource[1] =&gt; [[c_custkey, o_orderstatus, o_orderkey]]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fragment 1 [SOURCE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output layout: [c_custkey, o_orderstatus, o_orderkey]</td>
</tr>
<tr>
<td>Output partitioning: SINGLE [ ]</td>
</tr>
<tr>
<td>Stage Execution Strategy: UNGROUPED_EXECUTION</td>
</tr>
<tr>
<td>- CrossJoin =&gt; [[c_custkey, o_orderstatus, o_orderkey]]</td>
</tr>
<tr>
<td>Distribution: REPLICATED</td>
</tr>
<tr>
<td>- ScanFilter[table = awsdatacatalog:HiveTableHandle{schemaName=tpch100, tableName=customer, analyzePartitionValues=Optional.empty}, grouped = false, filterPredicate = (&quot;c_custkey&quot; = 123)] =&gt; [[c_custkey]]</td>
</tr>
<tr>
<td>LAYOUT: tpch100_customer</td>
</tr>
<tr>
<td>c_custkey := c_custkey:int:0:REGULAR</td>
</tr>
<tr>
<td>- LocalExchange[SINGLE] () =&gt; [[o_orderstatus, o_orderkey]]</td>
</tr>
<tr>
<td>- RemoteSource[2] =&gt; [[o_orderstatus, o_orderkey]]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fragment 2 [SOURCE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output layout: [o_orderstatus, o_orderkey]</td>
</tr>
<tr>
<td>Output partitioning: BROADCAST [ ]</td>
</tr>
<tr>
<td>Stage Execution Strategy: UNGROUPED_EXECUTION</td>
</tr>
<tr>
<td>- ScanFilterProject[table = awsdatacatalog:HiveTableHandle{schemaName=tpch100, tableName=orders, analyzePartitionValues=Optional.empty}, grouped = false, filterPredicate = (&quot;o_custkey&quot; = 123)] =&gt; [[o_orderstatus, o_orderkey]]</td>
</tr>
<tr>
<td>LAYOUT: tpch100_orders</td>
</tr>
</tbody>
</table>

707
The example query was optimized into a cross join for better performance. The results show that `tpch100.orders` will be distributed as the BROADCAST distribution type. This implies that the `tpch100.orders` table will be distributed to all nodes that perform the join operation. The BROADCAST distribution type will require that the all of the filtered results of the `tpch100.orders` table fit into the memory of each node that performs the join operation.

However, the `tpch100.customer` table is smaller than `tpch100.orders`. Because `tpch100.customer` requires less memory, you can rewrite the query to BROADCAST `tpch100.customer` instead of `tpch100.orders`. This reduces the chance of the query receiving the EXCEEDED_LOCAL_MEMORY_LIMIT error. This strategy assumes the following points:

- The `tpch100.customer.c_custkey` is unique in the `tpch100.customer` table.
- There is a one-to-many mapping relationship between `tpch100.customer` and `tpch100.orders`.

The following example shows the rewritten query.

```sql
SELECT
  c.c_custkey,
  o.o_orderkey,
  o.o_orderstatus
FROM tpch100.orders o
JOIN tpch100.customer c -- the filtered results of tpch100.customer are distributed to all nodes.
  ON c.c_custkey = o.o_custkey
WHERE c.c_custkey = 123
```

**EXPLAIN example 5: Use an EXPLAIN query to remove predicates that have no effect**

You can use an EXPLAIN query to check the effectiveness of filtering predicates. You can use the results to remove predicates that have no effect, as in the following example.

```sql
EXPLAIN
SELECT
  c.c_name
FROM tpch100.customer c
WHERE c.c_custkey = CAST(RANDOM() * 1000 AS INT)
  AND c.c_custkey BETWEEN 1000 AND 2000
  AND c.c_custkey = 1500
```

**Results**

Query Plan
- Output[c_name] => [[c_name]]
  - RemoteExchange[GATHER] => [[c_name]]
  - ScanFilterProject[table = awsdatacatalog:HiveTableHandle{schemaName=tpch100, tableName=customer, analyzePartitionValues=Optional.empty}, filterPredicate = (("c_custkey" = 1500) AND ("c_custkey" = CAST("random"() * 1E3 AS int))))] => [[c_name]]

LAYOUT: tpch100.customer
  c_custkey := c_custkey:int:0:REGULAR
  c_name := c_name:string:1:REGULAR
The `filterPredicate` in the results shows that the optimizer merged the original three predicates into two predicates and changed their order of application.

```sql
filterPredicate = ("c_custkey" = 1500) AND ("c_custkey" = CAST("random"() * 1E3 AS int))
```

Because the results show that the predicate `AND c.c_custkey BETWEEN 1000 AND 2000` has no effect, you can remove this predicate without changing the query results.

For information about the terms used in the results of EXPLAIN queries, see Understanding Athena EXPLAIN statement results (p. 712).

**EXPLAIN ANALYZE examples**

The following examples show example EXPLAIN ANALYZE queries and outputs.

**EXPLAIN ANALYZE example 1: Use EXPLAIN ANALYZE to show a query plan and computational cost in text format**

In the following example, EXPLAIN ANALYZE shows the execution plan and computational costs for a SELECT query on CloudFront logs. The format defaults to text output.

```sql
EXPLAIN ANALYZE SELECT FROM cloudfront_logs LIMIT 10
```

**Results**

<table>
<thead>
<tr>
<th>Fragment 1</th>
<th>CPU: 24.60ms, Input: 10 rows (1.48kB); per task: std.dev.: 0.00, Output: 10 rows (1.48kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Output layout: [date, time, location, bytes, requestip, method, host, uri, status, referrer,</td>
</tr>
<tr>
<td></td>
<td>os, browser, browserversion]</td>
</tr>
<tr>
<td></td>
<td>Limit[10] =&gt; [[date, time, location, bytes, requestip, method, host, uri, status, referrer,</td>
</tr>
<tr>
<td></td>
<td>os, \</td>
</tr>
<tr>
<td></td>
<td>browser, browserversion]]</td>
</tr>
<tr>
<td></td>
<td>CPU: 1.00ms (0.03%), Output: 10 rows (1.48kB)</td>
</tr>
<tr>
<td></td>
<td>Input avg.: 10.00 rows, Input std.dev.: 0.00%</td>
</tr>
<tr>
<td></td>
<td>LocalExchange[SINGLE] () =&gt; [[date, time, location, bytes, requestip, method, host, uri,</td>
</tr>
<tr>
<td></td>
<td>status, referrer, os, \</td>
</tr>
<tr>
<td></td>
<td>browser, browserversion]]</td>
</tr>
<tr>
<td></td>
<td>CPU: 0.00ms (0.00%), Output: 10 rows (1.48kB)</td>
</tr>
<tr>
<td></td>
<td>Input avg.: 0.63 rows, Input std.dev.: 387.30%</td>
</tr>
<tr>
<td></td>
<td>RemoteSource[2] =&gt; [[date, time, location, bytes, requestip, method, host, uri, status,</td>
</tr>
<tr>
<td></td>
<td>referrer, os, \</td>
</tr>
<tr>
<td></td>
<td>browser, browserversion]]</td>
</tr>
<tr>
<td></td>
<td>CPU: 1.00ms (0.03%), Output: 10 rows (1.48kB)</td>
</tr>
<tr>
<td></td>
<td>Input avg.: 0.63 rows, Input std.dev.: 387.30%</td>
</tr>
<tr>
<td>Fragment 2</td>
<td>CPU: 3.83s, Input: 998 rows (147.21kB); per task: std.dev.: 0.00, Output: 20 rows (2.95kB)</td>
</tr>
<tr>
<td></td>
<td>Output layout: [date, time, location, bytes, requestip, method, host, uri, status, referrer,</td>
</tr>
<tr>
<td></td>
<td>os, \</td>
</tr>
<tr>
<td></td>
<td>browser, browserversion]</td>
</tr>
<tr>
<td></td>
<td>LimitPartial[10] =&gt; [[date, time, location, bytes, requestip, method, host, uri, status,</td>
</tr>
<tr>
<td></td>
<td>referrer, os, \</td>
</tr>
<tr>
<td></td>
<td>browser, browserversion]]</td>
</tr>
<tr>
<td></td>
<td>CPU: 5.00ms (0.13%), Output: 20 rows (2.95kB)</td>
</tr>
<tr>
<td></td>
<td>Input avg.: 166.33 rows, Input std.dev.: 141.42%</td>
</tr>
</tbody>
</table>

TableScan[awsdatacatalog:HiveTableHandle(schemaName=default, tableName=cloudfront_logs,
analyzePartitionValues=Optional.empty,
grouped = false => [([date, time, location, bytes, requestip, method, host, uri, st
CPU: 3.82s (99.82%), Output: 998 rows (147.21kB)
  Input avg.: 166.33 rows, Input std.dev.: 141.42%
LAYOUT: default.cloudfront_logs
date := date:date:0:REGULAR
referrer := referrer:string:9:REGULAR
os := os:string:10:REGULAR
method := method:string:5:REGULAR
bytes := bytes:int:3:REGULAR
browser := browser:string:11:REGULAR
host := host:string:6:REGULAR
requestip := requestip:string:4:REGULAR
location := location:string:2:REGULAR
time := time:string:1:REGULAR
uri := uri:string:7:REGULAR
browserversion := browserversion:string:12:REGULAR
status := status:int:8:REGULAR

EXPLAIN ANALYZE example 2: Use EXPLAIN ANALYZE to show a query plan in
JSON format

The following example shows the execution plan and computational costs for a
SELECT query on CloudFront logs. The example specifies JSON as the output format.

EXPLAIN ANALYZE (FORMAT JSON) SELECT * FROM cloudfront_logs LIMIT 10

Results

```json
[
  "fragments": [{
    "id": "1",
    "stageStats": {
      "totalCpuTime": "3.31ms",
      "inputRows": "10 rows",
      "inputDataSize": "1514B",
      "stdDevInputRows": "0.00",
      "outputRows": "10 rows",
      "outputDataSize": "1514B"
    },
    "outputLayout": "date, time, location, bytes, requestip, method, host,\
    uri, status, referrer, os, browser, browserversion",
    "logicalPlan": {
      "1": {
        "name": "Limit",
        "identifier": "[10]",
        "outputs": ["date", "time", "location", "bytes", "requestip", "method",
        "host", "uri", "status", "referrer", "os", "browser", "browserversion"],
        "details": "",
        "distributedNodeStats": {
          "nodeCpuTime": "0.00ns",
          "nodeOutputRows": 10,
          "nodeOutputDataSize": "1514B",
          "operatorInputRowsStats": [{
            "nodeInputRows": 10.0,
            "nodeInputRowsStdDev": 0.0
          }]
        }
      },
      "children": [
```
EXPLAIN statement output types

EXPLAIN statement outputs can be one of two types:

- **Logical plan** – Shows the logical plan that the SQL engine uses to execute a statement. The syntax for this option is `EXPLAIN` or `EXPLAIN (TYPE LOGICAL)`.

Additional resources

For additional information, see the following resources.

- Understanding Athena EXPLAIN statement results (p. 712)
- Viewing execution plans for SQL queries (p. 340)
- Viewing statistics and execution details for completed queries (p. 357)
- Presto EXPLAIN documentation
- Presto EXPLAIN ANALYZE documentation

Understanding Athena EXPLAIN statement results

This topic provides a brief guide to the operational terms used in Athena EXPLAIN statement results.
**Distributed plan** – Shows an execution plan in a distributed environment. The output shows fragments, which are processing stages. Each plan fragment is processed by one or more nodes. Data can be exchanged between the nodes that process the fragments. The syntax for this option is `EXPLAIN (TYPE DISTRIBUTED)`.

In the output for a distributed plan, fragments (processing stages) are indicated by `Fragment number [fragment_type]`, where `number` is a zero-based integer and `fragment_type` specifies how the fragment is executed by the nodes. Fragment types, which provide insight into the layout of the data exchange, are described in the following table.

### Distributed plan fragment types

<table>
<thead>
<tr>
<th>Fragment type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SINGLE</td>
<td>The fragment is executed on a single node.</td>
</tr>
<tr>
<td>HASH</td>
<td>The fragment is executed on a fixed number of nodes. The input data is</td>
</tr>
<tr>
<td></td>
<td>distributed using a hash function.</td>
</tr>
<tr>
<td>ROUND_ROBIN</td>
<td>The fragment is executed on a fixed number of nodes. The input data is</td>
</tr>
<tr>
<td></td>
<td>distributed in a round-robin fashion.</td>
</tr>
<tr>
<td>BROADCAST</td>
<td>The fragment is executed on a fixed number of nodes. The input data is</td>
</tr>
<tr>
<td></td>
<td>broadcast to all nodes.</td>
</tr>
<tr>
<td>SOURCE</td>
<td>The fragment is executed on nodes where input splits are accessed.</td>
</tr>
</tbody>
</table>

**Exchange**

Exchange-related terms describe how data is exchanged between worker nodes. Transfers can be either local or remote.

**LocalExchange** `[exchange_type]`

Transfers data locally within worker nodes for different stages of a query. The value for `exchange_type` can be one of the logical or distributed exchange types as described later in this section.

**RemoteExchange** `[exchange_type]`

Transfers data between worker nodes for different stages of a query. The value for `exchange_type` can be one of the logical or distributed exchange types as described later in this section.

**Logical Exchange types**

The following exchange types describe actions taken during the exchange phase of a logical plan.

- **GATHER** – A single worker node gathers output from all other worker nodes. For example, the last stage of a select query gathers results from all nodes and writes the results to Amazon S3.
- **REPARTITION** – Sends the row data to a specific worker based on the partitioning scheme required to apply to the next operator.
- **REPLICATE** – Copies the row data to all workers.

**Distributed Exchange types**

The following exchange types indicate the layout of the data when they are exchanged between nodes in a distributed plan.
• **HASH** – The exchange distributes data to multiple destinations using a hash function.
• **SINGLE** – The exchange distributes data to a single destination.

## Scanning

The following terms describe how data is scanned during a query.

**TableScan**

Scans a table's source data from Amazon S3 or an Apache Hive connector and applies partition pruning generated from the filter predicate.

**ScanFilter**

Scans a table's source data from Amazon S3 or a Hive connector and applies partition pruning generated from the filter predicate and from additional filter predicates not applied through partition pruning.

**ScanFilterProject**

First, scans a table's source data from Amazon S3 or a Hive connector and applies partition pruning generated from the filter predicate and from additional filter predicates not applied through partition pruning. Then, modifies the memory layout of the output data into a new projection to improve performance of later stages.

## Join

Joins data between two tables. Joins can be categorized by join type and by distribution type.

### Join types

Join types define the way in which the join operation occurs.

**CrossJoin** – Produces the Cartesian product of the two tables joined.

**InnerJoin** – Selects records that have matching values in both tables.

**LeftJoin** – Selects all records from the left table and the matching records from the right table. If no match occurs, the result on the right side is NULL.

**RightJoin** – Selects all records from the right table, and the matching records from the left table. If no match occurs, the result on the left side is NULL.

**FullJoin** – Selects all records where there is a match in the left or right table records. The joined table contains all records from both the tables and fills in NULLs for missing matches on either side.

**Note**

For performance reasons, the query engine can rewrite a join query into a different join type to produce the same results. For example, an inner join query with predicate on one table can be rewritten into a CrossJoin. This pushes the predicate down to the scanning phase of the table so that fewer data are scanned.

### Join distribution types

Distribution types define how data is exchanged between worker nodes when the join operation is performed.

**Partitioned** – Both the left and right table are hash-partitioned across all worker nodes. Partitioned distribution consumes less memory in each node. Partitioned distribution can be much slower than replicated joins. Partitioned joins are suitable when you join two large tables.
Replicated – One table is hash-partitioned across all worker nodes and the other table is replicated to all worker nodes to perform the join operation. Replicated distribution can be much faster than partitioned joins, but it consumes more memory in each worker node. If the replicated table is too large, the worker node can experience an out-of-memory error. Replicated joins are suitable when one of the joined tables is small.

PREPARE

Creates a SQL statement with the name statement_name to be run at a later time. The statement can include parameters represented by question marks. To supply values for the parameters and run the prepared statement, use EXECUTE (p. 716).

Synopsis

PREPARE statement_name FROM statement

The following table describes the parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>statement_name</td>
<td>The name of the statement to be prepared. The name must be unique within the workgroup.</td>
</tr>
<tr>
<td>statement</td>
<td>A SELECT, CTAS, or INSERT INTO query.</td>
</tr>
</tbody>
</table>

Note

The maximum number of prepared statements in a workgroup is 1000.

Examples

The following example prepares a select query without parameters.

PREPARE my_select1 FROM SELECT * FROM nation

The following example prepares a select query that includes parameters. The values for productid and quantity will be supplied by the USING clause of an EXECUTE statement:

PREPARE my_select2 FROM SELECT order FROM orders WHERE productid = ? and quantity < ?

The following example prepares an insert query.

PREPARE my_insert FROM INSERT INTO cities_usa (city, state) SELECT city, state FROM cities_world WHERE country = ?

See also

Querying with prepared statements (p. 372)
EXECUTE (p. 716)
DEALLOCATE PREPARE (p. 716)
INSERT INTO (p. 697)

EXECUTE

Runs a prepared statement with the name statement_name. Parameter values for the question marks in the prepared statement are defined in the USING clause in a comma separated list. To create a prepared statement, use PREPARE (p. 715).

Synopsis

```
EXECUTE statement_name [ USING parameter1[, parameter2, ... ] ]
```

Examples

The following example prepares and executes a query with no parameters.

```
PREPARE my_select1 FROM
SELECT name FROM nation
EXECUTE my_select1
```

The following example prepares and executes a query with a single parameter.

```
PREPARE my_select2 FROM
SELECT * FROM "my_database"."my_table" WHERE year = ?
EXECUTE my_select2 USING 2012
```

This is equivalent to:

```
SELECT * FROM "my_database"."my_table" WHERE year = 2012
```

The following example prepares and executes a query with two parameters.

```
PREPARE my_select3 FROM
SELECT order FROM orders WHERE productid = ? and quantity < ?
EXECUTE my_select3 USING 346078, 12
```

See also

Querying with prepared statements (p. 372)
PREPARE (p. 715)
INSERT INTO (p. 697)

DEALLOCATE PREPARE

Removes the prepared statement with the specified name from the prepared statements in the current workgroup.
UNLOAD

Writes query results from a SELECT statement to the specified data format. Supported formats for UNLOAD include Apache Parquet, ORC, Apache Avro, and JSON. CSV is the only output format used by the Athena SELECT query, but you can use UNLOAD to write the output of a SELECT query to the formats that UNLOAD supports.

Although you can use the CTAS statement to output data in formats other than CSV, those statements also require the creation of a table in Athena. The UNLOAD statement is useful when you want to output the results of a SELECT query in a non-CSV format but do not require the associated table. For example, a downstream application might require the results of a SELECT query to be in JSON format, and Parquet or ORC might provide a performance advantage over CSV if you intend to use the results of the SELECT query for additional analysis.

Considerations and limitations

When you use the UNLOAD statement in Athena, keep in mind the following points:

- **No global ordering of files** – UNLOAD results are written to multiple files in parallel. If the SELECT query in the UNLOAD statement specifies a sort order, each file's contents are in sorted order, but the files are not sorted relative to each other.

- **Orphaned data not deleted** – In the case of a failure, Athena does not attempt to delete orphaned data. This behavior is the same as that for CTAS and INSERT INTO statements.

- **Maximum partitions** – The maximum number of partitions that can be used with UNLOAD is 100.

- **Metadata and manifest files** – Athena generates a metadata file and data manifest file for each UNLOAD query. The manifest tracks the files that the query wrote. Both files are saved to your Athena query result location in Amazon S3. For more information, see Identifying query output files (p. 352).

- **Encryption** – UNLOAD output files are encrypted according to the encryption configuration used for Amazon S3. To set up encryption configuration to encrypt your UNLOAD result, you can use the EncryptionConfiguration API.

- **Prepared statements** – UNLOAD can be used with prepared statements. For information about prepared statements in Athena, see Using parameterized queries (p. 370).

- **Service quotas** – UNLOAD uses DML query quotas. For quota information, see Service Quotas (p. 785).

- **Expected bucket owner** – The expected bucket owner setting does not apply to the destination Amazon S3 location specified in the UNLOAD query. The expected bucket owner setting applies only to
the Amazon S3 output location that you specify for Athena query results. For more information, see Specifying a query result location using the Athena console (p. 344).

Syntax

The UNLOAD statement uses the following syntax.

```
UNLOAD (SELECT col_name[, ...] FROM old_table)
TO 's3://my_athena_data_location/my_folder'/
WITH (property_name = 'expression'[, ...])
```

**Note**

The TO destination must specify a location in Amazon S3 that has no data. Before the UNLOAD query writes to the location specified, it verifies that the bucket location is empty. Because UNLOAD does not write data to the specified location if the location already has data in it, UNLOAD does not overwrite existing data. To reuse a bucket location as a destination for UNLOAD, delete the data in the bucket location, and then run the query again.

Parameters

Possible values for property_name are as follows.

- **format = 'file_format'**
  
  Required. Specifies the file format of the output. Possible values for file_format are ORC, PARQUET, AVRO, JSON, or TEXTFILE.

- **compression = 'compression_format'**
  
  Optional. This option is specific to the ORC and Parquet formats. For ORC, possible values are lz4, snappy, zlib, or zstd. For Parquet, possible values are gzip or snappy. For ORC, the default is zlib, and for Parquet, the default is gzip.

  **Note**

  This option does not apply to the AVRO format. Athena uses gzip for the JSON and TEXTFILE formats.

- **field_delimiter = 'delimiter'**
  
  Optional. Specifies a single-character field delimiter for files in CSV, TSV, and other text formats. The following example specifies a comma delimiter.

  ```
  WITH (field_delimiter = ',')
  ```

  Currently, multicharacter field delimiters are not supported. If you do not specify a field delimiter, the octal character \001 (^A) is used.

- **partitioned_by = ARRAY[col_name[...]]**
  
  Optional. An array list of columns by which the output is partitioned.

  **Note**

  In your SELECT statement, make sure that the names of the partitioned columns are last in your list of columns.

Examples

The following example writes the output of a SELECT query to the Amazon S3 location s3://DOC-EXAMPLE-BUCKET/unload_test_1/ using JSON format.
Functions

UNLOAD (SELECT * FROM old_table)
TO 's3://DOC-EXAMPLE-BUCKET/unload_test_1/
WITH (format = 'JSON')

The following example writes the output of a SELECT query in Parquet format using Snappy compression.

UNLOAD (SELECT * FROM old_table)
TO 's3://DOC-EXAMPLE-BUCKET/
WITH (format = 'PARQUET', compression = 'SNAPPY')

The following example writes four columns in text format, with the output partitioned by the last column.

UNLOAD (SELECT name1, address1, comment1, key1 FROM table1)
TO 's3://DOC-EXAMPLE-BUCKET/partitioned/
WITH (format = 'TEXTFILE', partitioned_by = ARRAY['key1'])

Functions in Amazon Athena

Athena supports some, but not all, Trino and Presto functions. For information, see Considerations and limitations (p. 764). For a list of the time zones that can be used with the AT TIME ZONE operator, see Supported time zones (p. 720).

Athena engine version 3

Functions in Athena engine version 3 are based on Trino. For information about Trino functions, operators, and expressions, see Functions and operators and the following subsections from the Trino documentation.

- Aggregate
- Array
- Binary
- Bitwise
- Color
- Comparison
- Conditional
- Conversion
- Date and time
- Decimal
- Geospatial
- HyperLogLog
- IP Address
- JSON
- Lambda
- Logical
- Machine learning
- Map
- Math
- Quantile digest
• Regular expression
• Session
• Set Digest
• String
• System
• Table
• Teradata
• T-Digest
• URL
• UUID
• Window

Athena engine version 2

Functions in Athena engine version 2 are based on Presto 0.217. For the geospatial functions in Athena engine version 2, see Geospatial functions in Athena engine version 2 (p. 404).

**Note**
Version-specific documentation for Presto 0.217 functions is no longer available. For information about current Presto functions, operators, and expressions, see Presto functions and operators, or visit the subcategory links in this section.

• Logical operators
• Comparison functions and operators
• Conditional expressions
• Conversion functions
• Mathematical functions and operators
• Bitwise functions
• Decimal functions and operators
• String functions and operators
• Binary functions
• Date and time functions and operators
• Regular expression functions
• JSON functions and operators
• URL functions
• Aggregate functions
• Window functions
• Color functions
• Array functions and operators
• Map functions and operators
• Lambda expressions and functions
• Teradata functions

**Supported time zones**

You can use the AT TIME ZONE operator in a SELECT timestamp statement to specify the timezone for the timestamp that is returned, as in the following example:
SELECT timestamp '2012-10-31 01:00 UTC' AT TIME ZONE 'America/Los_Angeles' AS la_time;

Results

<table>
<thead>
<tr>
<th>la_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012-10-30 18:00:00.000 America/Los_Angeles</td>
</tr>
</tbody>
</table>

The following list contains the time zones that can be used with the AT TIME ZONE operator in Athena. For additional timezone related functions and examples, see Timezone functions and examples (p. 729).

Africa/Abidjan  
Africa/Accra  
Africa/Addis_Ababa  
Africa/Algiers  
Africa/Asmara  
Africa/Asmera  
Africa/Bamako  
Africa/Bangui  
Africa/Banjul  
Africa/Bissau  
Africa/Blantyre  
Africa/Brazzaville  
Africa/Bujumbura  
Africa/Cairo  
Africa/Casablanca  
Africa/Ceuta  
Africa/Conakry  
Africa/Dakar  
Africa/Dar_es_Salaam  
Africa/Djibouti  
Africa/Douala  
Africa/El_Aaiun  
Africa/Freetown  
Africa/Gaborone  
Africa/Harare  
Africa/Johannesburg  
Africa/Juba  
Africa/Kampala  
Africa/Khartoum  
Africa/Kigali  
Africa/Kinshasa  
Africa/Lagos  
Africa/Libreville  
Africa/Lome  
Africa/Luanda  
Africa/Lubumbashi  
Africa/Lusaka  
Africa/Malabo  
Africa/Maputo  
Africa/Maseru  
Africa/Mbabane  
Africa/Mogadishu  
Africa/Monrovia  
Africa/Nairobi  
Africa/Ndjamena  
Africa/Niamey  
Africa/Nouakchott  
Africa/Ouagadougou  
Africa/Porto-Nov  
Africa/Sao_Tome
Supported time zones

Africa/Timbuktu
Africa/Tripoli
Africa/Tunis
Africa/Windhoek
America/Adak
America/Anchorage
America/Anguilla
America/Antigua
America/Argentina/Buenos_Aires
America/Argentina/Catamarca
America/Argentina/ComodRivadavia
America/Argentina/Cordoba
America/Argentina/Jujuy
America/Argentina/La_Rioja
America/Argentina/Mendoza
America/Argentina/Rio_Gallegos
America/Argentina/Salta
America/Argentina/San_Juan
America/Argentina/San_Luis
America/Argentina/Tucuman
America/Argentina/Ushuaia
America/Aruba
America/Asuncion
America/Atikokan
America/Atka
America/Bahia
America/Bahia_Banderas
America/Barbados
America/Belem
America/Belize
America/Blanc-Sablon
America/Boa_Vista
America/Bogota
America/Boise
America/Buenos_Aires
America/Cambridge_Bay
America/Campo_Grande
America/Cancun
America/Caracas
America/Catamarca
America/Cayenne
America/Cayman
America/Chicago
America/Chihuahua
America/Coral_Harbour
America/Cordoba
America/Costa_Rica
America/Creston
America/Cuiaba
America/Curacao
America/Danmarkshavn
America/Dawson
America/Dawson_Creek
America/Denver
America/Detroit
America/Dominica
America/Edmonton
America/Eirunepe
America/El_Salvador
America/Ensenada
America/Fort_Nelson
America/Fort_Wayne
America/Fortaleza
America/Glace_Bay
America/Godthab
Supported time zones

- America/Goose_Bay
- America/Grand_Turk
- America/Grenada
- America/Guadeloupe
- America/Guatemala
- America/Guayaquil
- America/Guyana
- America/Halifax
- America/Havana
- America/Hermosillo
- America/Indiana/Indianapolis
- America/Indiana/Knox
- America/Indiana/Marengo
- America/Indiana/Petersburg
- America/Indiana/Tell_City
- America/Indiana/Vevay
- America/Indiana/Vincennes
- America/Indiana/Winamac
- America/Indianapolis
- America/Inuvik
- America/Iqaluit
- America/Jamaica
- America/Jujuy
- America/Juneau
- America/Kentucky/Louisville
- America/Kentucky/Monticello
- America/Knox_IN
- America/Kralendijk
- America/La_Paz
- America/Lima
- America/Los_Angeles
- America/Louisville
- America/Lower_Princes
- America/Maceio
- America/Managua
- America/Manaus
- America/Marigot
- America/Martinique
- America/Matamoros
- America/Mazatlan
- America/Mendoza
- America/Menominee
- America/Merida
- America/Metlakatla
- America/Mexico_City
- America/Miquelon
- America/Moncton
- America/Monterrey
- America/Montevideo
- America/Montreal
- America/Montserrat
- America/Nassau
- America/New_York
- America/Nipigon
- America/Nome
- America/Noronha
- America/North_Dakota/Beulah
- America/North_Dakota/Center
- America/North_Dakota/New_Salem
- America/Ojinaga
- America/Panama
- America/Pangnirtung
- America/Paramaribo
- America/Phoenix
- America/Port-au-Prince
- America/Port_of_Spain
Supported time zones

- America/Porto_Acre
- America/Porto_Velho
- America/Puerto_Rico
- America/Punta_Arenas
- America/Rainy_River
- America/Regina
- America/Resolute
- America/Rio_Branco
- America/Rosario
- America/Santa_Isabel
- America/Santarem
- America/Santiago
- America/Santo_Domingo
- America/Sao_Paulo
- America/Scelesbysund
- America/Shiprock
- America/Sitka
- America/St_Barthelemy
- America/St_Johns
- America/St_Kitts
- America/St_Lucia
- America/St_Thomans
- America/St_Vincent
- America/Swift_Current
- America/Tegucigalpa
- America/Thule
- America/Thunder_Bay
- America/Tijuana
- America/Toronto
- America/Tortola
- America/Vancouver
- America/Virgin
- America/Whitehorse
- America/Winnipeg
- America/Yakutat
- America/Yellowknife
- Antarctica/Cape_Henlopen
- Antarctica/Davis
- Antarctica/DumontDUrville
- Antarctica/Macquarie
- Antarctica/Mawson
- Antarctica/Mcmurdo
- Antarctica/Palmer
- Antarctica/Rothera
- Antarctica/South_Pole
- Antarctica/Syowa
- Antarctica/Troll
- Antarctica/Vostok
- Arctic/Longyearbyen
- Asia/Aden
- Asia/Almaty
- Asia/Amman
- Asia/Anadyr
- Asia/Aqtion
- Asia/Aqtobe
- Asia/Ashgabat
- Asia/Ashkhabad
- Asia/Atyrau
- Asia/Baghdad
- Asia/Bahrain
- Asia/Baku
- Asia/Bangkok
- Asia/Barnaul
- Asia/Beirut
Supported time zones

Asia/Bishkek
Asia/Brunei
Asia/Calcutta
Asia/Chita
Asia/Choibalsan
Asia/Chongqing
Asia/Chungking
Asia/Colombo
Asia/Dacca
Asia/Damascus
Asia/Dhaka
Asia/Dili
Asia/Dubai
Asia/Dushanbe
Asia/Gaza
Asia/Harbin
Asia/Hebron
Asia/Ho_Chi Minh
Asia/Hong_Kong
Asia/Hovd
Asia/Irkutsk
Asia/Istanbul
Asia/Jakarta
Asia/Jayapura
Asia/Jerusalem
Asia/Kabul
Asia/Kamchatka
Asia/Karachi
Asia/Kashgar
Asia/Kathmandu
Asia/Katmandu
Asia/Khandyga
Asia/Kolkata
Asia/Krasnoyarsk
Asia/Kuala_Lumpur
Asia/Kuching
Asia/Kuwait
Asia/Macao
Asia/Macau
Asia/Magadan
Asia/Makassar
Asia/Manila
Asia/Muscat
Asia/Nicosia
Asia/Novokuznetsk
Asia/Novosibirsk
Asia/Omsk
Asia/Oral
Asia/Phnom_Penh
Asia/Pontianak
Asia/Pyongyang
Asia/Qatar
Asia/Qyzylorda
Asia/Rangoon
Asia/Riyadh
Asia/Reagan
Asia/Saigon
Asia/Sakhalin
Asia/Samarkand
Asia/Seoul
Asia/Shanghai
Asia/Singapore
Asia/Srednekolymsk
Asia/Taipei
Asia/Tashkent
Asia/Tbilisi
Asia/Tehran
Supported time zones

Asia/Tel_Aviv
Asia/Thimbu
Asia/Thimphu
Asia/Tokyo
Asia/Tomsk
Asia/Ujung_Pandang
Asia/Ulaanbaatar
Asia/Ulan_Bator
Asia/Urumqi
Asia/Ust-Nera
Asia/Vientiane
Asia/Vladivostok
Asia/Yakutsk
Asia/Yangon
Asia/Yekaterinburg
Asia/Yerevan
Atlantic/Azores
Atlantic/Bermuda
Atlantic/Canary
Atlantic/Cape_Verde
Atlantic/Faeroe
Atlantic/Fao
Atlantic/Faroe
Atlantic/Jan_Mayen
Atlantic/Madeira
Atlantic/Reykjavik
Atlantic/South_Georgia
Atlantic/St_Helena
Atlantic/Stanley
Australia/ACT
Australia/Adelaide
Australia/Brisbane
Australia/Broken_Hill
Australia/Canberra
Australia/Currie
Australia/Darwin
Australia/Eucla
Australia/Hobart
Australia/LHI
Australia/Lindeman
Australia/Lord_Howe
Australia/Melbourne
Australia/NSW
Australia/North
Australia/Perth
Australia/Queensland
Australia/South
Australia/Sydney
Australia/Tasmania
Australia/Victoria
Australia/West
Australia/Yancowinna
Brazil/Acre
Brazil/DeNoronha
Brazil/East
Brazil/West
CET
CST6CDT
Canada/Atlantic
Canada/Central
Canada/Eastern
Canada/Mountain
Canada/Newfoundland
Canada/Pacific
Canada/Saskatchewan
Canada/Yukon
Chile/Continental
Supported time zones

- Chile/EasterIsland
- Cuba
- EET
- EST5EDT
- Egypt
- Eire
- Europe/Amsterdam
- Europe/Andorra
- Europe/Astrakhan
- Europe/Athens
- Europe/Belfast
- Europe/Belgrade
- Europe/Berlin
- Europe/Bratislava
- Europe/Brussels
- Europe/Bucharest
- Europe/Budapest
- Europe/Busingen
- Europe/Chisinau
- Europe/Copenhagen
- Europe/Dublin
- Europe/Gibraltar
- Europe/Guernsey
- Europe/Helsinki
- Europe/Isle_of_Man
- Europe/Istanbul
- Europe/Jersey
- Europe/Kaliningrad
- Europe/Kiev
- Europe/Kirov
- Europe/Lisbon
- Europe/Ljubljana
- Europe/London
- Europe/Luxembourg
- Europe/Madrid
- Europe/Malta
- Europe/Mariehamn
- Europe/Minsk
- Europe/Monaco
- Europe/Moscow
- Europe/Nicosia
- Europe/Oslo
- Europe/Paris
- Europe/Podgorica
- Europe/Prague
- Europe/Riga
- Europe/Rome
- Europe/Samara
- Europe/San_Marino
- Europe/Sarajevo
- Europe/Simferopol
- Europe/Skopje
- Europe/Sofia
- Europe/Stockholm
- Europe/Tallinn
- Europe/Tirane
- Europe/Tiraspol
- Europe/Ulyanovsk
- Europe/Uzhgorod
- Europe/Vaduz
- Europe/Vatican
- Europe/Vienna
- Europe/Vilnius
- Europe/Volgograd
- Europe/Warsaw
- Europe/Zagreb
Supported time zones

Europe/Zaporozhye
Europe/Zurich
GB
GB-Eire
Hongkong
Iceland
Indian/Antananarivo
Indian/Chagos
Indian/Christmas
Indian/Cocos
Indian/Comoro
Indian/Kerguelen
Indian/Mahe
Indian/Maldives
Indian/Mauritius
Indian/Mayotte
Indian/Reunion
Iran
Israel
Jamaica
Japan
Kwajalein
Libya
MET
MST7MDT
Mexico/BajaNorte
Mexico/BajaSur
Mexico/General
NZ
NZ-CHAT
Navajo
PRC
PST8PDT
Pacific/Apia
Pacific/Auckland
Pacific/Bougainville
Pacific/Chatham
Pacific/Chuuk
Pacific/Easter
Pacific/Efate
Pacific/Enderbury
Pacific/Fakaofo
Pacific/Fiji
Pacific/Funafuti
Pacific/Galapagos
Pacific/Gambier
Pacific/Guadalcanal
Pacific/Guam
Pacific/Honolulu
Pacific/Johnston
Pacific/Kiritimati
Pacific/Kosrae
Pacific/Kwajalein
Pacific/Majuro
Pacific/Marquesas
Pacific/Midway
Pacific/Nauru
Pacific/Niue
Pacific/Norfolk
Pacific/Noumea
Pacific/Pago_Pago
Pacific/Palau
Pacific/Pitcairn
Pacific/Pohnpei
Pacific/Ponape
Pacific/Port_Moresby
Supported time zones

- Pacific/Rarotonga
- Pacific/Saipan
- Pacific/Samoa
- Pacific/Tahiti
- Pacific/Tarawa
- Pacific/Tongatapu
- Pacific/Truk
- Pacific/Wake
- Pacific/Wallis
- Pacific/Yap
- Poland
- Portugal
- ROK
- Singapore
- Turkey
- US/Alaska
- US/Aleutian
- US/Arizona
- US/Central
- US/East-Indiana
- US/Eastern
- US/Hawaii
- US/Indiana-Starke
- US/Michigan
- US/Mountain
- US/Pacific
- US/Pacific-New
- US/Samoa
- W-SU
- WET

Timezone functions and examples

Following are some additional timezone related functions and examples.

- **at_timezone**(timestamp, zone) – Returns the value of timestamp in the corresponding local time for zone.

  **Example**

  ```sql
  SELECT at_timezone(timestamp '2021-08-22 00:00 UTC', 'Canada/Newfoundland')
  ```

  **Result**

  `2021-08-21 21:30:00.000 Canada/Newfoundland`

- **timezone_hour**(timestamp) – Returns the hour of the time zone offset from timestamp as a bigint.

  **Example**

  ```sql
  SELECT timezone_hour(timestamp '2021-08-22 04:00 UTC' AT TIME ZONE 'Canada/Newfoundland')
  ```

  **Result**

  `-2`

- **timezone_minute**(timestamp) – Returns the minute of the time zone offset from timestamp as a bigint.

  **Example**
SELECT timezone_minute(timestamp '2021-08-22 04:00 UTC' AT TIME ZONE 'Canada/Newfoundland')

Result

-30

- with_timezone(timestamp, zone) – Returns a timestamp with time zone from the specified `timestamp` and `zone` values.

Example

SELECT with_timezone(timestamp '2021-08-22 04:00', 'Canada/Newfoundland')

Result

2021-08-22 04:00:00.000 Canada/Newfoundland

**DDL statements**

Use the following DDL statements directly in Athena.

The Athena query engine is based in part on HiveQL DDL.

Athena does not support all DDL statements, and there are some differences between HiveQL DDL and Athena DDL. For more information, see the reference topics in this section and Unsupported DDL (p. 731).

**Topics**

- Unsupported DDL (p. 731)
- ALTER DATABASE SET DBPROPERTIES (p. 732)
- ALTER TABLE ADD COLUMNS (p. 732)
- ALTER TABLE ADD PARTITION (p. 733)
- ALTER TABLE DROP PARTITION (p. 735)
- ALTER TABLE RENAME PARTITION (p. 735)
- ALTER TABLE REPLACE COLUMNS (p. 736)
- ALTER TABLE SET LOCATION (p. 737)
- ALTER TABLE SET TBLPROPERTIES (p. 738)
- CREATE DATABASE (p. 739)
- CREATE TABLE (p. 740)
- CREATE TABLE AS (p. 744)
- CREATE VIEW (p. 750)
- DESCRIBE (p. 751)
- DESCRIBE VIEW (p. 754)
- DROP DATABASE (p. 754)
- DROP TABLE (p. 755)
- DROP VIEW (p. 756)
- MSCK REPAIR TABLE (p. 756)
Unsupported DDL

The following DDL statements are not supported by Athena:

- ALTER INDEX
- ALTER TABLE `table_name` ARCHIVE PARTITION
- ALTER TABLE `table_name` CLUSTERED BY
- ALTER TABLE `table_name` EXCHANGE PARTITION
- ALTER TABLE `table_name` NOT CLUSTERED
- ALTER TABLE `table_name` NOT SKewed
- ALTER TABLE `table_name` NOT SORTED
- ALTER TABLE `table_name` NOT STORED AS DIRECTORIES
- ALTER TABLE `table_name` partitionSpec CHANGE COLUMNS
- ALTER TABLE `table_name` partitionSpec COMPACT
- ALTER TABLE `table_name` partitionSpec CONCATENATE
- ALTER TABLE `table_name` partitionSpec SET FILEFORMAT
- ALTER TABLE `table_name` RENAME TO
- ALTER TABLE `table_name` SET SERDEPROPERTIES
- ALTER TABLE `table_name` SET SKewed LOCATION
- ALTER TABLE `table_name` SKewed BY
- ALTER TABLE `table_name` TOUCH
- ALTER TABLE `table_name` UNARCHIVE PARTITION
- COMMIT
- CREATE INDEX
- CREATE ROLE
- CREATE TABLE `table_name` LIKE `existing_table_name`
- CREATE TEMPORARY MACRO
- DELETE FROM
- DESCRIBE DATABASE
- DFS
- DROP INDEX
- DROP ROLE
- DROP TEMPORARY MACRO
- EXPORT TABLE
- GRANT ROLE
- IMPORT TABLE
- LOCK DATABASE
ALTER DATABASE SET DBPROPERTIES

Creates one or more properties for a database. The use of DATABASE and SCHEMA are interchangeable; they mean the same thing.

Synopsis

```
ALTER (DATABASE|SCHEMA) database_name
SET DBPROPERTIES ('property_name'='property_value' [, ...) ]
```

Parameters

```
SET DBPROPERTIES ('property_name'='property_value' [, ...) ]
```

Specifies a property or properties for the database named `property_name` and establishes the value for each of the properties respectively as `property_value`. If `property_name` already exists, the old value is overwritten with `property_value`.

Examples

```
ALTER DATABASE jd_datasets
SET DBPROPERTIES ('creator'='John Doe', 'department'='applied mathematics');
```

```
ALTER SCHEMA jd_datasets
SET DBPROPERTIES ('creator'='Jane Doe');
```

ALTER TABLE ADD COLUMNS

Adds one or more columns to an existing table. When the optional PARTITION syntax is used, updates partition metadata.
### Synopsis

```sql
ALTER TABLE table_name
[PARTITION
    (partition_col1_name = partition_col1_value
     [,partition_col2_name = partition_col2_value][,...])
ADD COLUMNS (col_name data_type)
```

### Parameters

**PARTITION (partition_col_name = partition_col_value [...])**

Creates a partition with the column name/value combinations that you specify. Enclose `partition_col_value` in quotation marks only if the data type of the column is a string.

**ADD COLUMNS (col_name data_type [,col_name data_type,...])**

Adds columns after existing columns but before partition columns.

### Examples

```sql
ALTER TABLE events ADD COLUMNS (eventowner string)

ALTER TABLE events PARTITION (awsregion='us-west-2') ADD COLUMNS (event string)

ALTER TABLE events PARTITION (awsregion='us-west-2') ADD COLUMNS (eventdescription string)
```

### Notes

- To see a new table column in the Athena Query Editor navigation pane after you run `ALTER TABLE ADD COLUMNS`, manually refresh the table list in the editor, and then expand the table again.
- `ALTER TABLE ADD COLUMNS` does not work for columns with the `date` datatype. To workaround this issue, use the `timestamp` datatype instead.

### ALTER TABLE ADD PARTITION

Creates one or more partition columns for the table. Each partition consists of one or more distinct column name/value combinations. A separate data directory is created for each specified combination, which can improve query performance in some circumstances. Partitioned columns don’t exist within the table data itself, so if you use a column name that has the same name as a column in the table itself, you get an error. For more information, see [Partitioning data in Athena](p. 260).

In Athena, a table and its partitions must use the same data formats but their schemas may differ. For more information, see [Updates in tables with partitions](p. 384).

For information about the resource-level permissions required in IAM policies (including `glue:CreatePartition`), see [AWS Glue API permissions: Actions and resources reference](p. 540). For troubleshooting information about permissions when using Athena, see the Permissions (p. 772) section of the [Troubleshooting in Athena](p. 766) topic.
### Synopsis

| ALTER TABLE table_name ADD [IF NOT EXISTS] PARTITION (partition_col1_name = partition_col1_value [,partition_col2_name = partition_col2_value] [,...]) [LOCATION 'location1'] | PARTITION (partition_colA_name = partition_colA_value [,partition_colB_name = partition_colB_value] [,,...]) [LOCATION 'location2'] [,...]|  

### Parameters

When you add a partition, you specify one or more column name/value pairs for the partition and the Amazon S3 path where the data files for that partition reside.

**[IF NOT EXISTS]**

Causes the error to be suppressed if a partition with the same definition already exists.

**PARTITION (partition_col_name = partition_col_value [,...])**

Creates a partition with the column name/value combinations that you specify. Enclose `partition_col_value` in string characters only if the data type of the column is a string.

**[LOCATION 'location']**

Specifies the directory in which to store the partitions defined by the preceding statement.

### Examples

```
ALTER TABLE orders ADD
  PARTITION (dt = '2016-05-14', country = 'IN');

ALTER TABLE orders ADD
  PARTITION (dt = '2016-05-14', country = 'IN')
  PARTITION (dt = '2016-05-15', country = 'IN');

ALTER TABLE orders ADD
  PARTITION (dt = '2016-05-14', country = 'IN') LOCATION 's3://mystorage/path/to/INDIA_14_May_2016/'
  PARTITION (dt = '2016-05-15', country = 'IN') LOCATION 's3://mystorage/path/to/INDIA_15_May_2016/';

ALTER TABLE flights_parquet ADD IF NOT EXISTS
  PARTITION (year = '2021')
```

### Zero byte _$folder$ files

If you run an ALTER TABLE ADD PARTITION statement and mistakenly specify a partition that already exists and an incorrect Amazon S3 location, zero byte placeholder files of the format `partition_value_$folder$` are created in Amazon S3. You must remove these files manually.
To prevent this from happening, use the ADD IF NOT EXISTS syntax in your ALTER TABLE ADD PARTITION statement, like this:

```sql
ALTER TABLE table_name ADD IF NOT EXISTS PARTITION [...]```

### ALTER TABLE DROP PARTITION

Drops one or more specified partitions for the named table.

#### Synopsis

```sql
ALTER TABLE table_name DROP [IF EXISTS] PARTITION (partition_spec) [, PARTITION (partition_spec)]
```

#### Parameters

**[IF EXISTS]**

Suppresses the error message if the partition specified does not exist.

**PARTITION (partition_spec)**

Each `partition_spec` specifies a column name/value combination in the form `partition_col_name = partition_col_value [, ...].`

#### Examples

```sql
ALTER TABLE orders
DROP PARTITION (dt = '2014-05-14', country = 'IN');
```

```sql
ALTER TABLE orders
DROP PARTITION (dt = '2014-05-14', country = 'IN'), PARTITION (dt = '2014-05-15', country = 'IN');
```

#### Notes

The `ALTER TABLE DROP PARTITION` statement does not provide a single syntax for dropping all partitions at once or support filtering criteria to specify a range of partitions to drop.

As a workaround, you can use the AWS Glue API `GetPartitions` and `BatchDeletePartition` actions in scripting. The `GetPartitions` action supports complex filter expressions like those in a SQL WHERE expression. After you use `GetPartitions` to create a filtered list of partitions to delete, you can use the `BatchDeletePartition` action to delete the partitions in batches of 25.

### ALTER TABLE RENAME PARTITION

Renames a partition column, `partition_spec`, for the table named `table_name`, to `new_partition_spec`.

For information about partitioning, see [Partitioning data in Athena](p. 260).
ALTER TABLE REPLACE COLUMNS

Synopsis

ALTER TABLE table_name PARTITION (partition_spec) RENAME TO PARTITION (new_partition_spec)

Parameters

PARTITION (partition_spec)

Each partition_spec specifies a column name/value combination in the form partition_col_name = partition_col_value [, ...].

Examples

ALTER TABLE orders
PARTITION (dt = '2014-05-14', country = 'IN') RENAME TO PARTITION (dt = '2014-05-15', country = 'IN');

ALTER TABLE REPLACE COLUMNS

Removes all existing columns from a table created with the LazySimpleSerDe (p. 324) and replaces them with the set of columns specified. When the optional PARTITION syntax is used, updates partition metadata. You can also use ALTER TABLE REPLACE COLUMNS to drop columns by specifying only the columns that you want to keep.

Synopsis

ALTER TABLE table_name
[PARTITION (partition_col_name = partition_col_value [, partition_col2_name = partition_col2_value][,...])]
REPLACE COLUMNS (col_name data_type [, col_name data_type, ...])

Parameters

PARTITION (partition_col_name = partition_col_value [...])

Specifies a partition with the column name/value combinations that you specify. Enclose partition_col_value in quotation marks only if the data type of the column is a string.

REPLACE COLUMNS (col_name data_type [, col_name data_type, ...])

Replaces existing columns with the column names and datatypes specified.

Notes

- To see the change in table columns in the Athena Query Editor navigation pane after you run ALTER TABLE REPLACE COLUMNS, you might have to manually refresh the table list in the editor, and then expand the table again.
- ALTER TABLE REPLACE COLUMNS does not work for columns with the date datatype. To workaround this issue, use the timestamp datatype in the table instead.
- Note that even if you are replacing just a single column, the syntax must be ALTER TABLE table-name REPLACE COLUMNS, with columns in the plural. You must specify not only the column that you want to replace, but the columns that you want to keep – if not, the columns that you do not specify
will be dropped. This syntax and behavior derives from Apache Hive DDL. For reference, see *Add/Replace columns* in the Apache documentation.

**Example**

In the following example, the table names_cities, which was created using the *LazySimpleSerDe* (p. 324), has three columns named col1, col2, and col3. All columns are of type string. To show the columns in the table, the following command uses the *SHOW COLUMNS* (p. 758) statement.

```
SHOW COLUMNS IN names_cities
```

Result of the query:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>col1</td>
<td></td>
</tr>
<tr>
<td>col2</td>
<td></td>
</tr>
<tr>
<td>col3</td>
<td></td>
</tr>
</tbody>
</table>

The following *ALTER TABLE REPLACE COLUMNS* command replaces the column names with first_name, last_name, and city. The underlying source data is not affected.

```
ALTER TABLE names_cities
REPLACE COLUMNS (first_name string, last_name string, city string)
```

To test the result, *SHOW COLUMNS* is run again.

```
SHOW COLUMNS IN names_cities
```

Result of the query:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>first_name</td>
<td></td>
</tr>
<tr>
<td>last_name</td>
<td></td>
</tr>
<tr>
<td>city</td>
<td></td>
</tr>
</tbody>
</table>

Another way to show the new column names is to *preview the table* (p. 254) in the Athena Query Editor or run your own *SELECT* query.

**ALTER TABLE SET LOCATION**

Changes the location for the table named table_name, and optionally a partition with partition_spec.

**Synopsis**

```
ALTER TABLE table_name [ PARTITION (partition_spec) ] SET LOCATION 'new location'
```

**Parameters**

**PARTITION (partition_spec)**

Specifies the partition with parameters partition_spec whose location you want to change. The partition_spec specifies a column name/value combination in the form partition_col_name = partition_col_value.


**SET LOCATION 'new location'**

Specifies the new location, which must be an Amazon S3 location. For information about syntax, see Table Location in Amazon S3 (p. 257).

### Examples

```
ALTER TABLE customers PARTITION (zip='98040', state='WA') SET LOCATION 's3://mystorage/custdata/';
```

### ALTER TABLE SET TBLPROPERTIES

Adds custom or predefined metadata properties to a table and sets their assigned values. To see the properties in a table, use the SHOW TBLPROPERTIES (p. 763) command.

Apache Hive Managed tables are not supported, so setting 'EXTERNAL'='FALSE' has no effect.

### Synopsis

```
ALTER TABLE table_name SET TBLPROPERTIES ('property_name' = 'property_value' [ , ... ])
```

### Parameters

**SET TBLPROPERTIES ('property_name' = 'property_value' [ , ... ])**

Specifies the metadata properties to add as `property_name` and the value for each as `property_value`. If `property_name` already exists, its value is set to the newly specified `property_value`.

The following predefined table properties have special uses.

<table>
<thead>
<tr>
<th>Predefined property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>classification</td>
<td>Indicates the data type for AWS Glue. Possible values are csv, parquet, orc, avro, or json. For more information, see the TBLPROPERTIES section of CREATE TABLE (p. 740).</td>
</tr>
<tr>
<td>has_encrypted_data</td>
<td>Indicates whether the dataset specified by LOCATION is encrypted. For more information, see the TBLPROPERTIES section of CREATE TABLE (p. 740) and Creating tables based on encrypted datasets in Amazon S3 (p. 526).</td>
</tr>
<tr>
<td>orc.compress</td>
<td>Specifies a compression format for data in ORC format. For more information, see ORC SerDe (p. 332).</td>
</tr>
<tr>
<td>parquet.compression</td>
<td>Specifies a compression format for data in Parquet format. For more information, see Parquet SerDe (p. 334).</td>
</tr>
<tr>
<td>write.compression</td>
<td>Specifies a compression format for data in the textfile or JSON formats. For the Parquet and ORC formats, use the parquet.compression and orc.compress properties respectively.</td>
</tr>
<tr>
<td>projection.*</td>
<td>Custom properties used in partition projection that allow Athena to know what partition patterns to expect when it runs a query on a table. For more information, see Partition projection with Amazon Athena (p. 265).</td>
</tr>
</tbody>
</table>
CREATE DATABASE

Predefined property | Description
--- | ---
skip.header.line.count | Ignores headers in data when you define a table. For more information, see Ignoring headers (p. 324).
storage.location.template | Specifies a custom Amazon S3 path template for projected partitions. For more information, see Setting up partition projection (p. 267).

Example

ALTER TABLE orders
SET TBLPROPERTIES ('notes'="Please don’t drop this table.");

CREATE DATABASE

Creates a database. The use of DATABASE and SCHEMA is interchangeable. They mean the same thing.

**Note**

For an example of creating a database, creating a table, and running a SELECT query on the table in Athena, see Getting started (p. 9).

Synopsis

CREATE (DATABASE|SCHEMA) [IF NOT EXISTS] database_name
[COMMENT 'database_comment']
[LOCATION 'S3_loc']
[WITH DBPROPERTIES ('property_name' = 'property_value') [, ...]]

Parameters

**[IF NOT EXISTS]**

Causes the error to be suppressed if a database named database_name already exists.

**[COMMENT database_comment]**

Establishes the metadata value for the built-in metadata property named comment and the value you provide for database_comment. In AWS Glue, the COMMENT contents are written to the Description field of the database properties.

**[LOCATION S3_loc]**

Specifies the location where database files and metastore will exist as S3_loc. The location must be an Amazon S3 location.

**[WITH DBPROPERTIES ('property_name' = 'property_value') [, ...]]**

Allows you to specify custom metadata properties for the database definition.

Examples

CREATE DATABASE clickstreams;

CREATE DATABASE IF NOT EXISTS clickstreams
CREATE TABLE

COMMENT 'Site Foo clickstream data aggregates'
LOCATION 's3://myS3location/clickstreams/
WITH DBPROPERTIES ('creator'='Jane D.', 'Dept.'='Marketing analytics');

Viewing database properties

To view the database properties for a database that you create in AWSDataCatalog using CREATE
DATABASE, you can use the AWS CLI command `aws glue get-database`, as in the following example:

```
aws glue get-database --name <your-database-name>
```

In JSON output, the result looks like the following:

```
{
  "Database": {
    "Name": "<your-database-name>",
    "Description": "<your-database-comment>",
    "LocationUri": "s3://<your-database-location>",
    "Parameters": {
      "<your-database-property-name>": "<your-database-property-value>"
    },
    "CreateTime": 1603383451.0,
    "CreateTableDefaultPermissions": [
      {
        "Principal": {
          "DataLakePrincipalIdentifier": "IAM_ALLOWED_PRINCIPALS"
        },
        "Permissions": [
          "ALL"
        ]
      }
    ]
  }
}
```

For more information about the AWS CLI, see the AWS Command Line Interface User Guide.

CREATE TABLE

Creates a table with the name and the parameters that you specify.

**Note**

This page contains summary reference information. For more information about creating tables in Athena and an example CREATE TABLE statement, see Creating tables in Athena (p. 251). For an example of creating a database, creating a table, and running a SELECT query on the table in Athena, see Getting started (p. 9).

**Synopsis**

```
CREATE EXTERNAL TABLE [IF NOT EXISTS]
[db_name.]table_name [(col_name data_type [COMMENT col_comment] [, ... ])]
[COMMENT table_comment]
[PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]
[CLUSTERED BY (col_name, col_name, ...) INTO num_buckets BUCKETS]
[ROW FORMAT row_format]
[STORED AS file_format]
[WITH SERDEPROPERTIES (...)]
[LOCATION 's3://bucket_name/[folder]/']
```
CREATE TABLE

```sql
[TBLPROPERTIES ( ['has_encrypted_data'='true | false',]
['classification'='aws_glue_classification',] property_name=property_value [, ...] ) ]
```

### Parameters

**EXTERNAL**

Specifies that the table is based on an underlying data file that exists in Amazon S3, in the LOCATION that you specify. Except when creating governed tables (p. 500) or Iceberg (p. 504) tables, always use the EXTERNAL keyword. If you use CREATE TABLE without the EXTERNAL keyword for non-governed, non-Iceberg tables, Athena issues an error. When you create an external table, the data referenced must comply with the default format or the format that you specify with the ROW FORMAT, STORED AS, and WITH SERDEPROPERTIES clauses.

**[IF NOT EXISTS]**

Causes the error message to be suppressed if a table named `table_name` already exists.

**[db_name.]table_name**

Specifies a name for the table to be created. The optional `db_name` parameter specifies the database where the table exists. If omitted, the current database is assumed. If the table name includes numbers, enclose `table_name` in quotation marks, for example "table123". If `table_name` begins with an underscore, use backticks, for example, ` _mytable `. Special characters (other than underscore) are not supported.

Athena table names are case-insensitive; however, if you work with Apache Spark, Spark requires lowercase table names.

**[( col_name data_type [COMMENT col_comment] [, ...])]**

Specifies the name for each column to be created, along with the column's data type. Column names do not allow special characters other than underscore (_). If `col_name` begins with an underscore, enclose the column name in backticks, for example ` _mycolumn `.

The `data_type` value can be any of the following:

- **boolean** – Values are `true` and `false`.
- **tinyint** – A 8-bit signed integer in two's complement format, with a minimum value of `-2^7` and a maximum value of `2^7-1`.
- **smallint** – A 16-bit signed integer in two's complement format, with a minimum value of `-2^15` and a maximum value of `2^15-1`.
- **int** – In Data Definition Language (DDL) queries like `CREATE TABLE`, use the `int` keyword to represent an integer. In other queries, use the keyword `integer`, where `integer` is represented as a 32-bit signed value in two's complement format, with a minimum value of `-2^31` and a maximum value of `2^31-1`. In the JDBC driver, `integer` is returned, to ensure compatibility with business analytics applications.
- **bigint** – A 64-bit signed integer in two's complement format, with a minimum value of `-2^63` and a maximum value of `2^63-1`.
- **double** – A 64-bit signed double-precision floating point number. The range is 4.94065645841246544e-324 to 1.79769313486231570e+308, positive or negative. `double` follows the IEEE Standard for Floating-Point Arithmetic (IEEE 754).
- **float** – A 32-bit signed single-precision floating point number. The range is 1.40129846432481707e-45 to 3.40282346638528860e+38, positive or negative. `float` follows the IEEE Standard for Floating-Point Arithmetic (IEEE 754). Equivalent to the `real` in Presto. In Athena, use `float` in DDL statements like `CREATE TABLE` and `real` in SQL functions like `SELECT`
CAST. The AWS Glue crawler returns values in float, and Athena translates real and float types internally (see the June 5, 2018 (p. 831) release notes).

- **decimal** [ (precision, scale) ], where **precision** is the total number of digits, and **scale** (optional) is the number of digits in fractional part, the default is 0. For example, use these type definitions: decimal(11, 5), decimal(15). The maximum value for **precision** is 38, and the maximum value for **scale** is 38.

To specify decimal values as literals, such as when selecting rows with a specific decimal value in a query DDL expression, specify the decimal type definition, and list the decimal value as a literal (in single quotes) in your query, as in this example: decimal_value = decimal '0.12'.

- **char** – Fixed length character data, with a specified length between 1 and 255, such as char(10). For more information, see CHAR Hive data type.

- **varchar** – Variable length character data, with a specified length between 1 and 65535, such as varchar(10). For more information, see VARCHAR Hive data type.

- **string** – A string literal enclosed in single or double quotes.

  **Note**
  Non-string data types cannot be cast to string in Athena; cast them to varchar instead.

- **binary** – (for data in Parquet)

- **date** – A date in ISO format, such as YYYY-MM-DD. For example, date '2008-09-15'. An exception is the OpenCSVSerDe, which uses the number of days elapsed since January 1, 1970. For more information, see OpenCSVSerDe for processing CSV (p. 329).

- **timestamp** – Date and time instant in a java.sql.Timestamp compatible format up to a maximum resolution of milliseconds, such as yyyy-mm-dd HH:mm:ss[.f...]. For example, timestamp '2008-09-15 03:04:05.324'. An exception is the OpenCSVSerDe, which uses TIMESTAMP data in the UNIX numeric format (for example, 1579059880000). For more information, see OpenCSVSerDe for processing CSV (p. 329).

- **array** < data_type >

- **map** < primitive_type, data_type >

- **struct** < col_name : data_type [comment col_comment] [, ...] >

[**COMMENT table_comment**]

Creates the comment table property and populates it with the table_comment you specify.

[**PARTITIONED BY (col_name data_type [ COMMENT col_comment ], ...)**]

Creates a partitioned table with one or more partition columns that have the col_name, data_type and col_comment specified. A table can have one or more partitions, which consist of a distinct column name and value combination. A separate data directory is created for each specified combination, which can improve query performance in some circumstances. Partitioned columns don’t exist within the table data itself. If you use a value for col_name that is the same as a table column, you get an error. For more information, see Partitioning Data (p. 260).

**Note**
After you create a table with partitions, run a subsequent query that consists of the **MSCK REPAIR TABLE** (p. 756) clause to refresh partition metadata, for example, MSCK REPAIR TABLE cloudfront_logs; For partitions that are not Hive compatible, use **ALTER TABLE ADD PARTITION** (p. 733) to load the partitions so that you can query the data.

[**CLUSTERED BY (col_name, col_name, ...) INTO num_buckets BUCKETS**]

Divides, with or without partitioning, the data in the specified col_name columns into data subsets called buckets. The num_buckets parameter specifies the number of buckets to create. Bucketing can improve the performance of some queries on large data sets.
CREATE TABLE

[ROW FORMAT row_format]

Specifies the row format of the table and its underlying source data if applicable. For row_format, you can specify one or more delimiters with the DELIMITED clause or, alternatively, use the SERDE clause as described below. If ROW_FORMAT is omitted or ROW_FORMAT DELIMITED is specified, a native SerDe is used.

- [DELIMITED FIELDS TERMINATED BY char [ESCAPED BY char]]
- [DELIMITED COLLECTION ITEMS TERMINATED BY char]
- [MAP KEYS TERMINATED BY char]
- [LINES TERMINATED BY char]
- [NULL DEFINED AS char]

Available only with Hive 0.13 and when the STORED AS file format is TEXTFILE.

--OR--

- SERDE 'serde_name' [WITH SERDEPROPERTIES ("property_name" = "property_value", "property_name" = "property_value" [, ...])]

The serde_name indicates the SerDe to use. The WITH SERDEPROPERTIES clause allows you to provide one or more custom properties allowed by the SerDe.

[STORED AS file_format]

Specifies the file format for table data. If omitted, TEXTFILE is the default. Options for file_format are:

- SEQUENCEFILE
- TEXTFILE
- RCFILE
- ORC
- PARQUET
- AVRO
- ION
- INPUTFORMAT input_format_classname OUTPUTFORMAT output_format_classname

[LOCATION 's3://bucket_name/[folder]/']

Specifies the location of the underlying data in Amazon S3 from which the table is created. The location path must be a bucket name or a bucket name and one or more folders. If you are using partitions, specify the root of the partitioned data. For more information about table location, see Table location in Amazon S3 (p. 257). For information about data format and permissions, see Requirements for tables in Athena and data in Amazon S3 (p. 252).

Use a trailing slash for your folder or bucket. Do not use file names or glob characters.

Use:

s3://mybucket/
s3://mybucket/folder/
s3://mybucket/folder/anotherfolder/

Don't use:

s3://path_to_bucket
s3://path_to_bucket/*

s3://path_to_bucket/mydatafile.dat

[TBLPROPERTIES ( 'has_encrypted_data'='true | false', ]

 property_name=property_value [ , ... ] )]

Specifies custom metadata key-value pairs for the table definition in addition to predefined table properties, such as "comment".

Athena has a built-in property, `has_encrypted_data`. Set this property to `true` to indicate that the underlying dataset specified by `LOCATION` is encrypted. If omitted and if the workgroup's settings do not override client-side settings, `false` is assumed. If omitted or set to `false` when underlying data is encrypted, the query results in an error. For more information, see Encryption at rest (p. 521).

To run ETL jobs, AWS Glue requires that you create a table with the `classification` property to indicate the data type for AWS Glue as `csv`, `parquet`, `orc`, `avro`, or `json`. For example, `classification='csv'`. ETL jobs will fail if you do not specify this property. You can subsequently specify it using the AWS Glue console, API, or CLI. For more information, see Using AWS Glue jobs for ETL with Athena (p. 30) and Authoring Jobs in AWS Glue in the AWS Glue Developer Guide.

For more information about creating tables, see Creating tables in Athena (p. 251).

**CREATE TABLE AS**

Creates a new table populated with the results of a `SELECT` (p. 692) query. To create an empty table, use `CREATE TABLE` (p. 740). For additional information about `CREATE TABLE AS` beyond the scope of this reference topic, see Creating a table from query results (CTAS) (p. 281).

**Note**

For CTAS statements, the expected bucket owner setting does not apply to the destination table location in Amazon S3. The expected bucket owner setting applies only to the Amazon S3 output location that you specify for Athena query results. For more information, see Specifying a query result location using the Athena console (p. 344).

**Topics**

- Synopsis (p. 692)
- CTAS table properties (p. 745)
- Examples (p. 750)

**Synopsis**

```
CREATE TABLE table_name
[ WITH ( property_name = expression [, ... ] ) ]
AS query
[ WITH [ NO ] DATA ]
```

**Where:**

**WITH ( property_name = expression [, ... ] )**

A list of optional CTAS table properties, some of which are specific to the data storage format. See CTAS table properties (p. 745).
query

A SELECT (p. 692) query that is used to create a new table.

Important
If you plan to create a query with partitions, specify the names of partitioned columns last in the list of columns in the SELECT statement.

[ WITH [ NO ] DATA ]

If WITH NO DATA is used, a new empty table with the same schema as the original table is created.

Important
If you run a CTAS query that specifies an external_location in a workgroup that enforces a query results location (p. 638), the query fails with an error message. To see the query results location specified for the workgroup, see the workgroup's details (p. 643). If your workgroup overrides the client-side setting for query results location, Athena creates your table in the following location:

Note
To include column headers in your query result output, you can use a simple SELECT query instead of a CTAS query. You can retrieve the results from your query results location or download the results directly using the Athena console. For more information, see Working with query results, recent queries, and output files (p. 344).

CTAS table properties

Each CTAS table in Athena has a list of optional CTAS table properties that you specify using WITH (property_name = expression [, ...] ). For information about using these parameters, see Examples of CTAS queries (p. 286).

WITH (property_name = expression [, ...], )

  table_type = ['HIVE', 'ICEBERG']

  Optional. The default is HIVE. Specifies the table type of the resulting table

  Example:

  WITH (table_type = 'ICEBERG')

  external_location = [location]

  Note
  Because Iceberg tables are not external, this property does not apply to Iceberg tables. To define the root location of an Iceberg table in a CTAS statement, use the location property described later in this section.

  Optional. The location where Athena saves your CTAS query in Amazon S3.

  Example:

  WITH (external_location = 's3://DOC-EXAMPLE-BUCKET/tables/parquet_table/')

  Athena does not use the same path for query results twice. If you specify the location manually, make sure that the Amazon S3 location that you specify has no data. Athena never attempts to delete your data. If you want to use the same location again, manually delete the data, or your CTAS query will fail.

  If you run a CTAS query that specifies an external_location in a workgroup that enforces a query results location (p. 638), the query fails with an error message. To see the query results location specified for the workgroup, see the workgroup's details (p. 643).

  If your workgroup overrides the client-side setting for query results location, Athena creates your table in the following location:
If you do not use the `external_location` property to specify a location and your workgroup does not override client-side settings, Athena uses your client-side setting (p. 344) for the query results location to create your table in the following location:

```
s3://query-results-location-setting/Unsaved-or-query-name/year/month/date/tables/query-id/
```

**is_external = [boolean]**

Optional. Indicates if the table is an external table. The default is true. For Iceberg tables, this must be set to false.

Example:

```
WITH (is_external = false)
```

**location = [location]**

Required for Iceberg tables. Specifies the root location for the Iceberg table to be created from the query results.

Example:

```
WITH (location = 's3://DOC-EXAMPLE-BUCKET/tables/iceberg_table/')
```

**format = [storage_format]**

The storage format for the CTAS query results, such as ORC, PARQUET, AVRO, JSON, ION, or TEXTFILE. For Iceberg tables, the allowed formats are ORC, PARQUET, and AVRO. If omitted, PARQUET is used by default. The name of this parameter, `format`, must be listed in lowercase, or your CTAS query will fail.

Example:

```
WITH (format = 'PARQUET')
```

**partitioned_by = ARRAY[col_name[,...]]**

Note

This property does not apply to Iceberg tables. To use partition transforms for Iceberg tables, use the partitioning property described later in this section.

Optional. An array list of columns by which the CTAS table will be partitioned. Verify that the names of partitioned columns are listed last in the list of columns in the `SELECT` statement.

**partitioning = ARRAY[partition_transform, ...]**

Optional. Specifies the partitioning of the Iceberg table to be created. Iceberg supports a wide variety of partition transforms and partition evolution. Partition transforms are summarized in the following table.

<table>
<thead>
<tr>
<th>Transform</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year(ts)</td>
<td>Creates a partition for each year. The partition value is the integer difference in years between ts and January 1, 1970.</td>
</tr>
<tr>
<td>Transform</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>month(ts)</td>
<td>Creates a partition for each month of each year. The partition value is the integer difference in months between ( ts ) and January 1, 1970.</td>
</tr>
<tr>
<td>day(ts)</td>
<td>Creates a partition for each day of each year. The partition value is the integer difference in days between ( ts ) and January 1, 1970.</td>
</tr>
<tr>
<td>hour(ts)</td>
<td>Creates a partition for each hour of each day. The partition value is a timestamp with the minutes and seconds set to zero.</td>
</tr>
<tr>
<td>bucket( (x, nbuckets) )</td>
<td>Hashes the data into the specified number of buckets. The partition value is an integer hash of ( x ), with a value between 0 and ( nbuckets - 1 ), inclusive.</td>
</tr>
<tr>
<td>truncate( (s, nchars) )</td>
<td>Makes the partition value the first ( nchars ) characters of ( s ).</td>
</tr>
</tbody>
</table>

**Example:**

```
WITH (partitioning = ARRAY['month(order_date)', 'bucket(account_number, 10)', 'country'])
```

**bucketed_by = ARRAY[ column_name[...], bucket_count = [int] ]**

**Note**

This property does not apply to Iceberg tables. For Iceberg tables, use partitioning with bucket transform.

An array list of buckets to bucket data. If omitted, Athena does not bucket your data in this query.

**bucket_count = [int]**

**Note**

This property does not apply to Iceberg tables. For Iceberg tables, use partitioning with bucket transform.

The number of buckets for bucketing your data. If omitted, Athena does not bucket your data.

**Example:**

```sql
CREATE TABLE bucketed_table WITH (bucketed_by = ARRAY[\( column_name \)], bucket_count = 30, format = 'PARQUET', external_location = 's3:///DOC-EXAMPLE-BUCKET/tables/parquet_table/') AS SELECT * FROM table_name
```

**write_target_data_file_size_bytes = [long]**

Optional. Specifies the target size in bytes of the files produced by Athena. Defaults to 512 MB.

**Example:**

```sql
WITH (write_target_data_file_size_bytes = 536870912)
```
optimize_rewrite_min_data_file_size_bytes = [long]

Optional. Data optimization specific configuration. Files smaller than the specified value are included for optimization. The default is 0.75 times the value of write_target_data_file_size_bytes. For more information, see Optimizing Iceberg tables (p. 515).

Example:
WITH (optimize_rewrite_min_data_file_size_bytes = 402653184)

optimize_rewrite_max_data_file_size_bytes = [long]

Optional. Data optimization specific configuration. Files larger than the specified value are included for optimization. The default is 1.8 times the value of write_target_data_file_size_bytes. For more information, see Optimizing Iceberg tables (p. 515).

Example:
WITH (optimize_rewrite_max_data_file_size_bytes = 966367641)

optimize_rewrite_data_file_threshold = [int]

Optional. Data optimization specific configuration. If there are fewer data files that require optimization than the given threshold, the files are not rewritten. This allows the accumulation of more data files to produce files closer to the target size and skip unnecessary computation for cost savings. The default is 5. For more information, see Optimizing Iceberg tables (p. 515).

Example:
WITH (optimize_rewrite_data_file_threshold = 5)

optimize_rewrite_delete_file_threshold = [int]

Optional. Data optimization specific configuration. If there are fewer delete files associated with a data file than the threshold, the data file is not rewritten. This allows the accumulation of more delete files for each data file for cost savings. The default is 2. For more information, see Optimizing Iceberg tables (p. 515).

Example:
WITH (optimize_rewrite_delete_file_threshold = 2)

vacuum_min_snapshots_to_keep = [int]

Optional. Vacuum specific configuration. The minimum number of most recent snapshots to retain. The default is 1. For more information, see VACUUM (p. 702).

**Note**
The vacuum_min_snapshots_to_keep property requires Athena engine version 3.

Example:
WITH (vacuum_min_snapshots_to_keep = 1)
vacuum_max_snapshot_age_seconds = [long]

Optional. Vacuum specific configuration. A period in seconds that represents the age of the snapshots to retain. The default is 432000 (5 days). For more information, see VACUUM (p. 702).

Note
The vacuum_max_snapshot_age_seconds property requires Athena engine version 3.

Example:

```sql
WITH (vacuum_max_snapshot_age_seconds = 432000)
```

write_compression = [compression_format]

The compression type to use for any storage format that allows compression to be specified. The compression_format value specifies the compression to be used when the data is written to the table. You can specify compression for the TEXTFILE, JSON, PARQUET, and ORC file formats.

For example, if the format property specifies PARQUET as the storage format, the value for write_compression specifies the compression format for Parquet. In this case, specifying a value for write_compression is equivalent to specifying a value for parquet_compression.

Similarly, if the format property specifies ORC as the storage format, the value for write_compression specifies the compression format for ORC. In this case, specifying a value for write_compression is equivalent to specifying a value for orc_compression.

Multiple compression format table properties cannot be specified in the same CTAS query. For example, you cannot specify both write_compression and parquet_compression in the same query. For information about the compression types that are supported for each file format, see Athena compression support (p. 299).

orc_compression = [compression_format]

The compression type to use for the ORC file format when ORC data is written to the table. For example, WITH (orc_compression = 'ZLIB'). Chunks within the ORC file (except the ORC Postscript) are compressed using the compression that you specify. If omitted, ZLIB compression is used by default for ORC.

Note
For consistency, we recommend that you use the write_compression property instead of orc_compression. Use the format property to specify the storage format as ORC, and then use the write_compression property to specify the compression format that ORC will use.

parquet_compression = [compression_format]

The compression type to use for the Parquet file format when Parquet data is written to the table. For example, WITH (parquet_compression = 'SNAPPY'). This compression is applied to column chunks within the Parquet files. If omitted, GZIP compression is used by default for Parquet.

Note
For consistency, we recommend that you use the write_compression property instead of parquet_compression. Use the format property to specify the storage format as PARQUET, and then use the write_compression property to specify the compression format that PARQUET will use.
**field_delimiter = [delimiter]**

Optional and specific to text-based data storage formats. The single-character field delimiter for files in CSV, TSV, and text files. For example, WITH (field_delimiter = ','). Currently, multicharacter field delimiters are not supported for CTAS queries. If you don't specify a field delimiter, \001 is used by default.

**Examples**

For examples of CTAS queries, consult the following resources.

- Examples of CTAS queries (p. 286)
- Using CTAS and INSERT INTO for ETL and data analysis (p. 290)
- Use CTAS statements with Amazon Athena to reduce cost and improve performance
- Using CTAS and INSERT INTO to create a table with more than 100 partitions (p. 295)

**CREATE VIEW**

Creates a new view from a specified SELECT query. The view is a logical table that can be referenced by future queries. Views do not contain any data and do not write data. Instead, the query specified by the view runs each time you reference the view by another query.

**Note**

This topic provides summary information for reference. For more detailed information about using views in Athena, see Working with views (p. 361).

**Synopsis**

CREATE [ OR REPLACE ] VIEW view_name AS query

The optional OR REPLACE clause lets you update the existing view by replacing it. For more information, see Creating views (p. 365).

**Examples**

To create a view test from the table orders, use a query similar to the following:

```
CREATE VIEW test AS
SELECT
  orderkey,
  orderstatus,
  totalprice / 2 AS half
FROM orders;
```

To create a view orders_by_date from the table orders, use the following query:

```
CREATE VIEW orders_by_date AS
SELECT orderdate, sum(totalprice) AS price
FROM orders
GROUP BY orderdate;
```

To update an existing view, use an example similar to the following:
CREATE OR REPLACE VIEW test AS
SELECT orderkey, orderstatus, totalprice / 4 AS quarter
FROM orders;

See also SHOW COLUMNS (p. 758), SHOW CREATE VIEW (p. 760), DESCRIBE VIEW (p. 754), and DROP VIEW (p. 756).

DESCRIBE

Shows one or more columns, including partition columns, for the specified table. This command is useful for examining the attributes of complex columns.

Synopsis

DESCRIBE [EXTENDED | FORMATTED] [db_name.]table_name [PARTITION partition_spec] [col_name ( [.field_name] | ['$elem$'] | ['$key$'] | ['$value$'] )]

Important

The syntax for this statement is DESCRIBE table_name, not DESCRIBE TABLE table_name. Using the latter syntax results in the error message FAILED: SemanticException [Error 10001]: Table not found table.

Parameters

[EXTENDED | FORMATTED]

Determines the format of the output. Omitting these parameters shows column names and their corresponding data types, including partition columns, in tabular format. Specifying FORMATTED not only shows column names and data types in tabular format, but also detailed table and storage information. EXTENDED shows column and data type information in tabular format, and detailed metadata for the table in Thrift serialized form. This format is less readable and is useful primarily for debugging.

[PARTITION partition_spec]

If included, lists the metadata for the partition specified by partition_spec, where partition_spec is in the format (partition_column = partition_col_value, partition_column = partition_col_value, ...).

[ col_name ( [.field_name] | ['$elem$'] | ['$key$'] | ['$value$'] ) ]

Specifies the column and attributes to examine. You can specify .field_name for an element of a struct, '$elem$' for array element, '$key$' for a map key, and '$value$' for map value. You can specify this recursively to further explore the complex column.

Examples

DESCRIBE orders

DESCRIBE FORMATTED mydatabase.mytable PARTITION (part_col = 100) columnA;

The following query and output shows column and data type information from an impressions table based on Amazon EMR sample data.
### DESCRIBE impressions

<table>
<thead>
<tr>
<th>col_name</th>
<th>data_type</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestbegintime</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>adid</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>impressionid</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>referrer</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>useragent</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>usercookie</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>ip</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>number</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>processid</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>browsercookie</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>requestendtime</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>timers</td>
<td>struct<a href="">modellookup:string,requesttime:string</a></td>
<td>from deserializer</td>
</tr>
<tr>
<td>threadid</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>hostname</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>sessionid</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>dt</td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>

#### Partition Information

<table>
<thead>
<tr>
<th>col_name</th>
<th>data_type</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>dt</td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>

The following example query and output show the result for the same table when the FORMATTED option is used.

### DESCRIBE FORMATTED impressions

<table>
<thead>
<tr>
<th>col_name</th>
<th>data_type</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestbegintime</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>adid</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>impressionid</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>referrer</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>useragent</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>usercookie</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>ip</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>number</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>processid</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>browsercookie</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>requestendtime</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>timers</td>
<td>struct<a href="">modellookup:string,requesttime:string</a></td>
<td>from deserializer</td>
</tr>
<tr>
<td>threadid</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>hostname</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>sessionid</td>
<td>string</td>
<td>from deserializer</td>
</tr>
<tr>
<td>dt</td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>

#### Partition Information
Amazon Athena User Guide

DESCRIBE

# Detailed Table Information
Database: sampledb
Owner: hadoop
CreateTime: Thu Apr 23 02:55:21 UTC 2020
LastAccessTime: UNKNOWN
Protect Mode: None
Retention: 0
Location: s3://us-east-1.elasticmapreduce/samples/hive-ads/tables/impressions
Table Type: EXTERNAL_TABLE
Table Parameters:
  EXTERNAL TRUE
transient_lastDdlTime 1587610521

# Storage Information
SerDe Library: org.openx.data.jsonserde.JsonSerDe
InputFormat: org.apache.hadoop.mapred.TextInputFormat
OutputFormat: org.apache.hadoop.hive.ql.io.IgnoreKeyTextOutputFormat
Compressed: No
Num Buckets: -1
Bucket Columns: []
Sort Columns: []
Storage Desc Params:
  paths requestbegintime, adid, impressionid,
  referrer, useragent, usercookie, ip
  serialization.format 1

The following example query and output show the result for the same table when the EXTENDED option is used. The detailed table information is output on a single line, but is formatted here for readability.

DESCRIBE EXTENDED impressions

requestbegintime    string    from deserializer
adid                string    from deserializer
impressionid        string    from deserializer
referrer            string    from deserializer
useragent           string    from deserializer
usercookie          string    from deserializer
ip                   string    from deserializer
number               string    from deserializer
processid           string    from deserializer
browsercokie        string    from deserializer
requestendtime      string    from deserializer

timers struct<modellookup:string,requesttime:string> from deserializer
threadid            string    from deserializer
hostname            string    from deserializer
sessionid           string    from deserializer
dt                   string

# Partition Information

# Detailed Table Information
Table(tableName:impressions, dbName:sampledb, owner:hadoop, createTime:1587610521, lastAccessTime:0, retention:0, sd:StorageDescriptor(cols:
  [FieldSchema(name:requestbegintime, type:string, comment:null),

753

DESCRIBE VIEW

Shows the list of columns for the named view. This allows you to examine the attributes of a complex view.

Synopsis

DESCRIBE [view_name]

Example

DESCRIBE orders;

See also SHOW COLUMNS (p. 758), SHOW CREATE VIEW (p. 760), SHOW VIEWS (p. 764), and DROP VIEW (p. 756).

DROP DATABASE

Removes the named database from the catalog. If the database contains tables, you must either drop the tables before running DROP DATABASE or use the CASCADE clause. The use of DATABASE and SCHEMA are interchangeable. They mean the same thing.

Synopsis

DROP {DATABASE | SCHEMA} [IF EXISTS] database_name [RESTRICT | CASCADE]
**Parameters**

**[IF EXISTS]**

Causes the error to be suppressed if database_name doesn't exist.

**[RESTRICT|CASCADE]**

Determines how tables within database_name are regarded during the DROP operation. If you specify RESTRICT, the database is not dropped if it contains tables. This is the default behavior. Specifying CASCADE causes the database and all its tables to be dropped.

**Examples**

```
DROP DATABASE clickstreams;
```

```
DROP SCHEMA IF EXISTS clickstreams CASCADE;
```

**Note**

When you try to drop a database whose name has special characters (for example, `my-database`), you may receive an error message. To resolve this issue, try enclosing the database name in backtick (``) characters. For information about naming databases in Athena, see Names for tables, databases, and columns (p. 254).

**DROP TABLE**

Removes the metadata table definition for the table named table_name. When you drop an external table, the underlying data remains intact.

**Synopsis**

```
DROP TABLE [IF EXISTS] table_name
```

**Parameters**

**[ IF EXISTS ]**

Causes the error to be suppressed if table_name doesn't exist.

**Examples**

```
DROP TABLE fulfilled_orders
```

```
DROP TABLE IF EXISTS fulfilled_orders
```

When using the Athena console query editor to drop a table that has special characters other than the underscore (_), use backticks, as in the following example.

```
DROP TABLE `my-athena-database-01.my-athena-table`
```
When using the JDBC connector to drop a table that has special characters, backtick characters are not required.

```
DROP TABLE my-athena-database-01.my-athena-table
```

**DROP VIEW**

Drops (deletes) an existing view. The optional IF EXISTS clause causes the error to be suppressed if the view does not exist.

For more information, see Working with views (p. 361).

**Synopsis**

```
DROP VIEW [ IF EXISTS ] view_name
```

**Examples**

```
DROP VIEW orders_by_date

DROP VIEW IF EXISTS orders_by_date
```

See also CREATE VIEW (p. 750), SHOW COLUMNS (p. 758), SHOW CREATE VIEW (p. 760), SHOW VIEWS (p. 764), and DESCRIBE VIEW (p. 754).

**MSCK REPAIR TABLE**

Use the MSCK REPAIR TABLE command to update the metadata in the catalog after you add Hive compatible partitions.

The MSCK REPAIR TABLE command scans a file system such as Amazon S3 for Hive compatible partitions that were added to the file system after the table was created. MSCK REPAIR TABLE compares the partitions in the table metadata and the partitions in S3. If new partitions are present in the S3 location that you specified when you created the table, it adds those partitions to the metadata and to the Athena table.

When you add physical partitions, the metadata in the catalog becomes inconsistent with the layout of the data in the file system, and information about the new partitions needs to be added to the catalog. To update the metadata, run MSCK REPAIR TABLE so that you can query the data in the new partitions from Athena.

**Note**

MSCK REPAIR TABLE only adds partitions to metadata; it does not remove them. To remove partitions from metadata after the partitions have been manually deleted in Amazon S3, run the command ALTER TABLE `table-name` DROP PARTITION. For more information see ALTER TABLE DROP PARTITION (p. 735).

**Considerations and limitations**

When using MSCK REPAIR TABLE, keep in mind the following points:

- It is possible it will take some time to add all partitions. If this operation times out, it will be in an incomplete state where only a few partitions are added to the catalog. You should run MSCK REPAIR TABLE again.
TABLE on the same table until all partitions are added. For more information, see Partitioning data in Athena (p. 260).

- For partitions that are not compatible with Hive, use ALTER TABLE ADD PARTITION (p. 733) to load the partitions so that you can query their data.

- Partition locations to be used with Athena must use the s3 protocol (for example, s3://bucket/folder/). In Athena, locations that use other protocols (for example, s3a://bucket/folder/) will result in query failures when MSCK REPAIR TABLE queries are run on the containing tables.

- Because MSCK REPAIR TABLE scans both a folder its subfolders to find a matching partition scheme, be sure to keep data for separate tables in separate folder hierarchies. For example, suppose you have data for table A in s3://table-a-data and data for table B in s3://table-a-data/table-b-data. If both tables are partitioned by string, MSCK REPAIR TABLE will add the partitions for table B to table A. To avoid this, use separate folder structures like s3://table-a-data and s3://table-b-data instead. Note that this behavior is consistent with Amazon EMR and Apache Hive.

- Due to a known issue, MSCK REPAIR TABLE fails silently when partition values contain a colon (:) character (for example, when the partition value is a timestamp). As a workaround, use ALTER TABLE ADD PARTITION (p. 733).

### Synopsis

| MSCK REPAIR TABLE table_name |

### Examples

| MSCK REPAIR TABLE orders; |

### Troubleshooting

After you run MSCK REPAIR TABLE, if Athena does not add the partitions to the table in the AWS Glue Data Catalog, check the following:

- Make sure that the AWS Identity and Access Management (IAM) user or role has a policy that allows the glue:BatchCreatePartition action.

- Make sure that the IAM user or role has a policy with sufficient permissions to access Amazon S3, including the s3:DescribeJob action. For an example of which Amazon S3 actions to allow, see the example bucket policy in Cross-account access in Athena to Amazon S3 buckets (p. 537).

- Make sure that the Amazon S3 path is in lower case instead of camel case (for example, userId instead of userId).

- **Query timeouts** – MSCK REPAIR TABLE is best used when creating a table for the first time or when there is uncertainty about parity between data and partition metadata. If you use MSCK REPAIR TABLE to add new partitions frequently (for example, on a daily basis) and are experiencing query timeouts, consider using ALTER TABLE ADD PARTITION (p. 733).

- **Partitions missing from filesystem** – If you delete a partition manually in Amazon S3 and then run MSCK REPAIR TABLE, you may receive the error message Partitions missing from filesystem. This occurs because MSCK REPAIR TABLE doesn't remove stale partitions from table metadata. To remove the deleted partitions from table metadata, run ALTER TABLE DROP PARTITION (p. 735) instead. Note that SHOW PARTITIONS (p. 760) similarly lists only the partitions in metadata, not the partitions in the file system.

- **"NullPointerException name is null" error**

  If you use the AWS Glue CreateTable API operation or the AWS CloudFormation AWS::Glue::Table template to create a table for use in Athena without specifying the TableType property and then run
SHOW COLUMNS

a DDL query like SHOW CREATE TABLE or MSCK REPAIR TABLE, you can receive the error message FAILED: NullPointerException Name is null.

To resolve the error, specify a value for the TableInput TableType attribute as part of the AWS Glue CreateTable API call or AWS CloudFormation template. Possible values for TableType include EXTERNAL_TABLE or VIRTUAL_VIEW.

This requirement applies only when you create a table using the AWS Glue CreateTable API operation or the AWS::Glue::Table template. If you create a table for Athena by using a DDL statement or an AWS Glue crawler, the TableType property is defined for you automatically.

The following sections provide some additional detail.

Allow glue:BatchCreatePartition in the IAM policy

Review the IAM policies attached to the user or role that you're using to run MSCK REPAIR TABLE. When you use the AWS Glue Data Catalog with Athena (p. 21), the IAM policy must allow the glue:BatchCreatePartition action. For an example of an IAM policy that allows the glue:BatchCreatePartition action, see AWS managed policy: AmazonAthenaFullAccess (p. 531).

Change the Amazon S3 path to lower case

The Amazon S3 path must be in lower case. If the S3 path is in camel case, MSCK REPAIR TABLE doesn't add the partitions to the AWS Glue Data Catalog. For example, if your S3 path is userId, the following partitions aren't added to the AWS Glue Data Catalog:

<table>
<thead>
<tr>
<th>S3 Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>s3://bucket/path/userId=1/</td>
</tr>
<tr>
<td>s3://bucket/path/userId=2/</td>
</tr>
<tr>
<td>s3://bucket/path/userId=3/</td>
</tr>
</tbody>
</table>

To resolve this issue, use flat case instead of camel case:

<table>
<thead>
<tr>
<th>S3 Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>s3://bucket/path/userid=1/</td>
</tr>
<tr>
<td>s3://bucket/path/userid=2/</td>
</tr>
<tr>
<td>s3://bucket/path/userid=3/</td>
</tr>
</tbody>
</table>

SHOW COLUMNS

Shows the column names for a specified table or view. To view additional column metadata (such as data type), see Listing or searching columns for a specified table or view (p. 445). To view all columns for all tables in all databases in AwsDataCatalog or for all tables in a specific database in AwsDataCatalog, see Listing all columns for all tables (p. 445) in the Querying AWS Glue Data Catalog (p. 442) topic.

Synopsis

```
SHOW COLUMNS (FROM|IN) database_name.table_name

SHOW COLUMNS (FROM|IN) table_name [(FROM|IN) database_name]
```

The FROM and IN keywords can be used interchangeably. If table_name or database_name has special characters like hyphens, surround the name with backquotes (for example, `my-database`.

```
show create table

Table `name`). Do not surround the `table_name` or `database_name` with single or double quotes. Currently, the use of LIKE and pattern matching expressions is not supported.

**Examples**

The following equivalent examples show the columns from the `orders` table in the `customers` database. The first two examples assume that `customers` is the current database.

```
SHOW COLUMNS FROM orders
SHOW COLUMNS IN orders
SHOW COLUMNS FROM customers.orders
SHOW COLUMNS IN customers.orders
SHOW COLUMNS FROM orders FROM customers
SHOW COLUMNS IN orders IN customers
```

**SHOW CREATE TABLE**

Analyzes an existing table named `table_name` to generate the query that created it.

**Synopsis**

```
SHOW CREATE TABLE [db_name.]table_name
```

**Parameters**

`TABLE [db_name.]table_name`

The `db_name` parameter is optional. If omitted, the context defaults to the current database.

**Note**
The table name is required.

**Examples**

```
SHOW CREATE TABLE orderclickstoday;
SHOW CREATE TABLE `salesdata.orderclickstoday`;
```

**Troubleshooting**

If you use the AWS Glue `CreateTable` API operation or the AWS CloudFormation `AWS::Glue::Table` template to create a table for use in Athena without specifying the `TableType` property and then run a DDL query like `SHOW CREATE TABLE` or `MSCK REPAIR TABLE`, you can receive the error message `FAILED: NullPointer Exception Name is null.`
To resolve the error, specify a value for the `TableInput` `TableType` attribute as part of the AWS Glue `CreateTable` API call or AWS CloudFormation template. Possible values for `TableType` include `EXTERNAL_TABLE` or `VIRTUAL_VIEW`.

This requirement applies only when you create a table using the AWS Glue `CreateTable` API operation or the AWS::Glue::Table template. If you create a table for Athena by using a DDL statement or an AWS Glue crawler, the `TableType` property is defined for you automatically.

**SHOW CREATE VIEW**

Shows the SQL statement that creates the specified view.

**Synopsis**

```
SHOW CREATE VIEW view_name
```

**Examples**

```
SHOW CREATE VIEW orders_by_date
```

See also CREATE VIEW (p. 750) and DROP VIEW (p. 756).

**SHOW DATABASES**

Lists all databases defined in the metastore. You can use DATABASES or SCHEMAS. They mean the same thing.

**Synopsis**

```
SHOW {DATABASES | SCHEMAS} [LIKE 'regular_expression']
```

**Parameters**

[LIKE 'regular_expression']

Filters the list of databases to those that match the `regular_expression` that you specify. For wildcard character matching, you can use the combination `.*`, which matches any character zero to unlimited times.

**Examples**

```
SHOW SCHEMAS;
```

```
SHOW DATABASES LIKE '.*analytics';
```

**SHOW PARTITIONS**

Lists all the partitions in an Athena table in unsorted order.
Synopsis

SHOW PARTITIONS table_name

- To show the partitions in a table and list them in a specific order, see the Listing partitions for a specific table (p. 444) section on the Querying AWS Glue Data Catalog (p. 442) page.
- To view the contents of a partition, see the Query the data (p. 262) section on the Partitioning data in Athena (p. 260) page.
- SHOW PARTITIONS does not list partitions that are projected by Athena but not registered in the AWS Glue catalog. For information about partition projection, see Partition projection with Amazon Athena (p. 265).
- SHOW PARTITIONS lists the partitions in metadata, not the partitions in the actual file system. To update the metadata after you delete partitions manually in Amazon S3, run ALTER TABLE DROP PARTITION (p. 735).

Examples

The following example query shows the partitions for the flight_delays_csv table, which shows flight table data from the US Department of Transportation. For more information about the example flight_delays_csv table, see LazySimpleSerDe for CSV, TSV, and custom-delimited files (p. 324). The table is partitioned by year.

SHOW PARTITIONS flight_delays_csv

Results

<table>
<thead>
<tr>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>2007</td>
</tr>
<tr>
<td>2015</td>
</tr>
<tr>
<td>1999</td>
</tr>
<tr>
<td>1993</td>
</tr>
<tr>
<td>1991</td>
</tr>
<tr>
<td>2003</td>
</tr>
<tr>
<td>1996</td>
</tr>
<tr>
<td>2014</td>
</tr>
<tr>
<td>2004</td>
</tr>
<tr>
<td>2011</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

The following example query shows the partitions for the impressions table, which contains sample web browsing data. For more information about the example impressions table, see Partitioning data in Athena (p. 260). The table is partitioned by the dt (datetime) column.

SHOW PARTITIONS impressions

Results

<table>
<thead>
<tr>
<th>dt</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009-04-12-16-00</td>
</tr>
<tr>
<td>2009-04-13-18-15</td>
</tr>
<tr>
<td>2009-04-14-00-20</td>
</tr>
<tr>
<td>2009-04-12-13-00</td>
</tr>
<tr>
<td>2009-04-13-02-15</td>
</tr>
<tr>
<td>2009-04-14-12-05</td>
</tr>
<tr>
<td>2009-04-14-06-15</td>
</tr>
<tr>
<td>2009-04-12-21-15</td>
</tr>
<tr>
<td>2009-04-15-22-15</td>
</tr>
</tbody>
</table>
Listing partitions in sorted order

To order the partitions in the results list, use the following SELECT syntax instead of SHOW PARTITIONS.

```sql
SELECT * FROM "table_name$partitions" ORDER BY column_name
```

The following query shows the list of partitions for the flight_delays_csv example, but in sorted order.

```sql
SELECT * FROM "flight_delays_csv$partitions" ORDER BY year
```

Results

<table>
<thead>
<tr>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1987</td>
</tr>
<tr>
<td>1988</td>
</tr>
<tr>
<td>1989</td>
</tr>
<tr>
<td>1990</td>
</tr>
<tr>
<td>1991</td>
</tr>
<tr>
<td>1992</td>
</tr>
<tr>
<td>1993</td>
</tr>
<tr>
<td>1994</td>
</tr>
<tr>
<td>1995</td>
</tr>
<tr>
<td>1996</td>
</tr>
<tr>
<td>1997</td>
</tr>
<tr>
<td>1998</td>
</tr>
<tr>
<td>1999</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

For more information, see the Listing partitions for a specific table (p. 444) section on the Querying AWS Glue Data Catalog (p. 442) page.

SHOW TABLES

Lists all the base tables and views in a database.

Synopsis

```
SHOW TABLES [IN database_name] ['regular_expression']
```

Parameters

[IN database_name]

Specifies the database_name from which tables will be listed. If omitted, the database from the current context is assumed.

**Note**

SHOW TABLES may fail if database_name uses an unsupported character (p. 254) such as a hyphen. As a workaround, try enclosing the database name in backticks.

['regular_expression']

Filters the list of tables to those that match the regular_expression you specify. To indicate any character in AWSDataCatalog tables, you can use the * or .* wildcard expression. For Apache
Hive databases, use the ".*" wildcard expression. To indicate a choice between characters, use the | character.

**Examples**

**Example – Show all of the tables in the database sampledb**

```sql
SHOW TABLES IN sampledb
```

**Results**

```sql
alb_logs
cloudfront_logs
elb_logs
flights_2016
flights_parquet
view_2016_flights_dfw
```

**Example – Show the names of all tables in sampledb that include the word "flights"**

```sql
SHOW TABLES IN sampledb '*flights*'
```

**Results**

```sql
flights_2016
flights_parquet
view_2016_flights_dfw
```

**Example – Show the names of all tables in sampledb that end in the word "logs"**

```sql
SHOW TABLES IN sampledb '*logs'
```

**Results**

```sql
alb_logs
cloudfront_logs
elb_logs
```

**SHOW TBLPROPERTIES**

Lists table properties for the named table.

**Synopsis**

```sql
SHOW TBLPROPERTIES table_name [(\'property_name\')]
```

**Parameters**

```sql
[(\'property_name\')]
```

If included, only the value of the property named property_name is listed.
SHOW VIEWS

Lists the views in the specified database, or in the current database if you omit the database name. Use the optional LIKE clause with a regular expression to restrict the list of view names. Athena returns a list of STRING type values where each value is a view name.

Synopsis

SHOW VIEWS [IN database_name] LIKE ['regular_expression']

Parameters

[IN database_name]
Specifies the database_name from which views will be listed. If omitted, the database from the current context is assumed.

[LIKE 'regular_expression']
Filters the list of views to those that match the regular_expression you specify. Only the wildcard *, which indicates any character, or |, which indicates a choice between characters, can be used.

Examples

SHOW VIEWS;

SHOW VIEWS IN marketing_analytics LIKE 'orders*';

See also SHOW COLUMNS (p. 758), SHOW CREATE VIEW (p. 760), DESCRIBE VIEW (p. 754), and DROP VIEW (p. 756).

Considerations and limitations for SQL queries in Amazon Athena

When running queries in Athena, keep in mind the following considerations and limitations:

- **Stored procedures** – Stored procedures are not supported.
- **Maximum number of partitions** – The maximum number of partitions you can create with CREATE TABLE AS SELECT (CTAS) statements is 100. For information, see CREATE TABLE AS (p. 744). For a workaround, see Using CTAS and INSERT INTO to create a table with more than 100 partitions (p. 295).
• **Unsupported statements** – The following statements are not supported:
  - CREATE TABLE LIKE is not supported.
  - DESCRIBE INPUT and DESCRIBE OUTPUT is not supported.
  - MERGE statements are not supported.
  - UPDATE statements are not supported.

• **Trino and Presto connectors** – Neither Trino nor Presto connectors are supported. Use Amazon Athena Federated Query to connect data sources. For more information, see *Using Amazon Athena Federated Query* (p. 60).

• **Timeouts on tables with many partitions** – Athena may time out when querying a table that has many thousands of partitions. This can happen when the table has many partitions that are not of type string. When you use type string, Athena prunes partitions at the metastore level. However, when you use other data types, Athena prunes partitions on the server side. The more partitions you have, the longer this process takes and the more likely your queries are to time out. To resolve this issue, set your partition type to string so that Athena prunes partitions at the metastore level. This reduces overhead and prevents queries from timing out.

• **Amazon S3 Glacier storage** – Athena does not support querying the data in the S3 Glacier flexible retrieval or S3 Glacier Deep Archive storage classes, or in the Archive access or deep archive access tiers of the S3 Intelligent Tiering storage class. Objects in the S3 Glacier Flexible Retrieval or S3 Glacier Deep Archive storage classes are ignored.

Data that is moved or transitioned to one of these classes are no longer readable or queryable by Athena even after storage class objects are restored. To make the restored objects that you want to query readable by Athena, copy the restored objects back into Amazon S3 to change their storage class. Alternatively, you can use the Amazon S3 Glacier Instant Retrieval storage class, which is queryable by Athena. For more information, see *Amazon S3 Glacier instant retrieval storage class*.

• **Files treated as hidden** – Athena treats source files that start with an underscore (__) or a dot (.) as hidden. To work around this limitation, rename the files.

• **Row or column size limitation** – The size of a single row or its columns cannot exceed 32 megabytes. This limit can be exceeded when, for example, a row in a CSV or JSON file contains a single column of 100 megabytes. Exceeding this limit can also produce the error message Line too long in text file. To work around this limitation, make sure that the sum of the data of the columns in any row is less than 32MB.

• **LIMIT clause maximum** – The maximum number of rows that can be specified for the LIMIT clause is 2147483647. Exceeding this limit results in the error message NOT_SUPPORTED: ORDER BY LIMIT > 2147483647 is not supported.

• **information_schema** – Querying information_schema is most performant if you have a small to moderate amount of AWS Glue metadata. If you have a large amount of metadata, errors can occur. For information about querying the information_schema database for AWS Glue metadata, see *Querying AWS Glue Data Catalog* (p. 442).

• **Array initializations** – Due to a limitation in Java, it is not possible to initialize an array in Athena that has more than 254 arguments.

For information about maximum query string length, quotas for query timeouts, and quotas for the active number of DML queries, see *Service Quotas* (p. 785).
Troubleshooting in Athena

The Athena team has gathered the following troubleshooting information from customer issues. Although not comprehensive, it includes advice regarding some common performance, timeout, and out of memory issues.

Topics

- CREATE TABLE AS SELECT (CTAS) (p. 766)
- Data file issues (p. 766)
- Federated queries (p. 768)
- JSON related errors (p. 769)
- MSCK REPAIR TABLE (p. 770)
- Output issues (p. 770)
- Partitioning issues (p. 771)
- Permissions (p. 772)
- Query syntax issues (p. 774)
- Throttling issues (p. 775)
- Views (p. 776)
- Workgroups (p. 776)
- Additional resources (p. 776)
- Athena error catalog (p. 777)

CREATE TABLE AS SELECT (CTAS)

Duplicated data occurs with concurrent CTAS statements

Athena does not maintain concurrent validation for CTAS. Make sure that there is no duplicate CTAS statement for the same location at the same time. Even if a CTAS or INSERT INTO statement fails, orphaned data can be left in the data location specified in the statement.

HIVE_TOO_MANY_OPEN_PARTITIONS

When you use a CTAS statement to create a table with more than 100 partitions, you may receive the error HIVE_TOO_MANY_OPEN_PARTITIONS: Exceeded limit of 100 open writers for partitions/buckets. To work around this limitation, you can use a CTAS statement and a series of INSERT INTO statements that create or insert up to 100 partitions each. For more information, see Using CTAS and INSERT INTO to create a table with more than 100 partitions (p. 295).

Data file issues

Athena cannot read hidden files

Athena treats sources files that start with an underscore (_) or a dot (.) as hidden. To work around this limitation, rename the files.
Athena cannot read files stored in Amazon S3 Glacier

Athena does not support querying the data in the S3 Glacier flexible retrieval or S3 Glacier Deep Archive storage classes. Objects in the S3 Glacier Flexible Retrieval and S3 Glacier Deep Archive storage classes are ignored. Data that is moved or transitioned to one of these classes are no longer readable or queryable by Athena even after storage class objects are restored. To make the restored objects that you want to query readable by Athena, copy the restored objects back into Amazon S3 to change their storage class, or use the Amazon S3 Glacier Instant Retrieval storage class instead, which is queryable by Athena. For more information, see Amazon S3 Glacier instant retrieval storage class.

Athena reads files that I excluded from the AWS Glue crawler

Athena does not recognize exclude patterns that you specify an AWS Glue crawler. For example, if you have an Amazon S3 bucket that contains both .csv and .json files and you exclude the .json files from the crawler, Athena queries both groups of files. To avoid this, place the files that you want to exclude in a different location.

HIVE_BAD_DATA: Error parsing field value

This error can occur in the following scenarios:

- The data type defined in the table doesn't match the source data, or a single field contains different types of data. For suggested resolutions, see My Amazon Athena query fails with the error "HIVE_BAD_DATA: Error parsing field value for field x: For input string: "12312845691" in the AWS Knowledge Center.

- Null values are present in an integer field. One workaround is to create the column with the null values as string and then use CAST to convert the field in a query, supplying a default value of 0 for nulls. For more information, see When I query CSV data in Athena, I get the error "HIVE_BAD_DATA: Error parsing field value " for field x: For input string: "" in the AWS Knowledge Center.

HIVE_CANNOT_OPEN_SPLIT: Error opening Hive split s3://bucket-name

This error can occur when you query an Amazon S3 bucket prefix that has a large number of objects. For more information, see How do I resolve the "HIVE_CANNOT_OPEN_SPLIT: Error opening Hive split s3://awsdoc-example-bucket/: Slow down" error in Athena? in the AWS Knowledge Center.

HIVE_CURSOR_ERROR:
com.amazonaws.services.s3.model.AmazonS3Exception:
The specified key does not exist

This error usually occurs when a file is removed when a query is running. Either rerun the query, or check your workflow to see if another job or process is modifying the files when the query is running.
HIVE_CURSOR_ERROR: Unexpected end of input stream

This message indicates the file is either corrupted or empty. Check the integrity of the file and rerun the query.

HIVE_FILESYSTEM_ERROR: Incorrect file size 1234567 for file

This message can occur when a file has changed between query planning and query execution. It usually occurs when a file on Amazon S3 is replaced in-place (for example, a PUT is performed on a key where an object already exists). Athena does not support deleting or replacing the contents of a file when a query is running. To avoid this error, schedule jobs that overwrite or delete files at times when queries do not run, or only write data to new files or partitions.

HIVE_UNKNOWN_ERROR: Unable to create input format

This error can be a result of issues like the following:

- The AWS Glue crawler wasn't able to classify the data format
- Certain AWS Glue table definition properties are empty
- Athena doesn't support the data format of the files in Amazon S3

For more information, see How do I resolve the error "unable to create input format" in Athena? in the AWS Knowledge Center or watch the Knowledge Center video.

org.apache.parquet.io.GroupColumnIO cannot be cast to org.apache.parquet.io.PrimitiveColumnIO

This error is caused by a parquet schema mismatch. A column that has a non-primitive type (for example, array) has been declared as a primitive type (for example, string) in AWS Glue. To troubleshoot this issue, check the data schema in the files and compare it with schema declared in AWS Glue.

The S3 location provided to save your query results is invalid.

Make sure that you have specified a valid S3 location for your query results. For more information, see Specifying a query result location (p. 344) in the Working with query results, recent queries, and output files (p. 344) topic.

Federated queries

For information about troubleshooting federated queries, see Common Problems in the awslabs/aws-athena-query-federation section of GitHub.
NULL or incorrect data errors when trying to read JSON data

NULL or incorrect data errors when you try read JSON data can be due to a number of causes. To identify lines that are causing errors when you are using the OpenX SerDe, set `ignore.malformed.json` to `true`. Malformed records will return as NULL. For more information, see I get errors when I try to read JSON data in Amazon Athena in the AWS Knowledge Center or watch the Knowledge Center video.

HIVE_BAD_DATA: Error parsing field value for field 0: java.lang.String cannot be cast to org.openx.data.jsonserde.json.JSONObject

The OpenX JSON SerDe (p. 321) throws this error when it fails to parse a column in an Athena query. This can happen if you define a column as a map or struct, but the underlying data is actually a string, int, or other primitive type.

HIVE_CURSOR_ERROR: Row is not a valid JSON object - JSONException: Duplicate key

This error occurs when you use Athena to query AWS Config resources that have multiple tags with the same name in different case. The solution is to run `CREATE TABLE` using `WITH SERDEPROPERTIES 'case.insensitive'='false'` and map the names. For information about `case.insensitive` and mapping, see JSON SerDe libraries (p. 319). For more information, see How do I resolve "HIVE_CURSOR_ERROR: Row is not a valid JSON object - JSONException: Duplicate key" when reading files from AWS Config in Athena? in the AWS Knowledge Center.

HIVE_CURSOR_ERROR messages with pretty-printed JSON

The Hive JSON SerDe (p. 320) and OpenX JSON SerDe (p. 321) libraries expect each JSON document to be on a single line of text with no line termination characters separating the fields in the record. If the JSON text is in pretty print format, you may receive an error message like HIVE_CURSOR_ERROR: Row is not a valid JSON Object or HIVE_CURSOR_ERROR: JsonParseException: Unexpected end-of-input: expected close marker for OBJECT when you attempt to query the table after you create it. For more information, see JSON data files in the OpenX SerDe documentation on GitHub.

Multiple JSON records return a SELECT COUNT of 1

If you're using the OpenX JSON SerDe (p. 321), make sure that the records are separated by a newline character. For more information, see The SELECT COUNT query in Amazon Athena returns only one record even though the input JSON file has multiple records in the AWS Knowledge Center.
Cannot query a table created by a AWS Glue crawler that uses a custom JSON classifier

The Athena engine does not support custom JSON classifiers. To work around this issue, create a new table without the custom classifier. To transform the JSON, you can use CTAS or create a view. For example, if you are working with arrays, you can use the UNNEST option to flatten the JSON. Another option is to use a AWS Glue ETL job that supports the custom classifier, convert the data to parquet in Amazon S3, and then query it in Athena.

MSCK REPAIR TABLE

For information about MSCK REPAIR TABLE related issues, see the Considerations and limitations (p. 756) and Troubleshooting (p. 757) sections of the MSCK REPAIR TABLE (p. 756) page.

Output issues

Unable to verify/create output bucket

This error can occur if the specified query result location doesn't exist or if the proper permissions are not present. For more information, see How do I resolve the "unable to verify/create output bucket" error in Amazon Athena? in the AWS Knowledge Center.

TIMESTAMP result is empty

Athena requires the Java TIMESTAMP format. For more information, see When I query a table in Amazon Athena, the TIMESTAMP result is empty in the AWS Knowledge Center.

Store Athena query output in a format other than CSV

By default, Athena outputs files in CSV format only. To output the results of a SELECT query in a different format, you can use the UNLOAD statement. For more information, see UNLOAD (p. 717). You can also use a CTAS query that uses the format table property (p. 745) to configure the output format. Unlike UNLOAD, the CTAS technique requires the creation of a table. For more information, see How can I store an Athena query output in a format other than CSV, such as a compressed format? in the AWS Knowledge Center.

The S3 location provided to save your query results is invalid

You can receive this error message if your output bucket location is not in the same Region as the Region in which you run your query. To avoid this, specify a query results location in the Region in which you run the query. For steps, see Specifying a query result location (p. 344).
Partitioning issues

MSCK REPAIR TABLE does not remove stale partitions

If you delete a partition manually in Amazon S3 and then run MSCK REPAIR TABLE, you may receive the error message Partitions missing from filesystem. This occurs because MSCK REPAIR TABLE doesn't remove stale partitions from table metadata. Use ALTER TABLE DROP PARTITION (p. 735) to remove the stale partitions manually. For more information, see the "Troubleshooting" section of the MSCK REPAIR TABLE (p. 756) topic.

MSCK REPAIR TABLE failure

When a large amount of partitions (for example, more than 100,000) are associated with a particular table, MSCK REPAIR TABLE can fail due to memory limitations. To work around this limit, use ALTER TABLE ADD PARTITION (p. 733) instead.

MSCK REPAIR TABLE detects partitions but doesn't add them to AWS Glue

This issue can occur if an Amazon S3 path is in camel case instead of lower case or an IAM policy doesn't allow the glue:BatchCreatePartition action. For more information, see MSCK REPAIR TABLE detects partitions in Athena but does not add them to the AWS Glue Data Catalog in the AWS Knowledge Center.

Partition projection ranges with the date format of dd-MM-yyyy-HH-mm-ss or yyyy-MM-dd do not work

To work correctly, the date format must be set to yyyy-MM-dd HH:00:00. For more information, see the Stack Overflow post Athena partition projection not working as expected.

PARTITION BY doesn't support the BIGINT type

Convert the data type to string and retry.

No meaningful partitions available

This error message usually means the partition settings have been corrupted. To resolve this issue, drop the table and create a table with new partitions.

Partition projection does not work in conjunction with range partitions

Check that the time range unit projection.<co1umnName>.interval.unit (p. 273) matches the delimiter for the partitions. For example, if partitions are delimited by days, then a range unit of hours will not work.
HIVE_UNKNOWN_ERROR: Unable to create input format

One or more of the glue partitions are declared in a different format as each glue partition has their own specific input format independently. Please check how your partitions are defined in AWS Glue.

HIVE_PARTITION_SCHEMA_MISMATCH

If the schema of a partition differs from the schema of the table, a query can fail with the error message HIVE_PARTITION_SCHEMA_MISMATCH. For more information, see Syncing partition schema to avoid "HIVE_PARTITION_SCHEMA_MISMATCH" (p. 27).

SemanticException table is not partitioned but partition spec exists

This error can occur when no partitions were defined in the CREATE TABLE statement. For more information, see How can I troubleshoot the error "FAILED: SemanticException table is not partitioned but partition spec exists" in Athena? in the AWS Knowledge Center.

Zero byte _$folder$ files

If you run an ALTER TABLE ADD PARTITION statement and mistakenly specify a partition that already exists and an incorrect Amazon S3 location, zero byte placeholder files of the format partition_value_$folder$ are created in Amazon S3. You must remove these files manually.

To prevent this from happening, use the ADD IF NOT EXISTS syntax in your ALTER TABLE ADD PARTITION statement, like this:

```
ALTER TABLE table_name ADD IF NOT EXISTS PARTITION [...]
```

Zero records returned from partitioned data

This issue can occur for a variety of reasons. For possible causes and resolutions, see I created a table in Amazon Athena with defined partitions, but when I query the table, zero records are returned in the AWS Knowledge Center.

See also HIVE_TOO_MANY_OPEN_PARTITIONS (p. 766).

Permissions

Access denied error when querying Amazon S3

This can occur when you don't have permission to read the data in the bucket, permission to write to the results bucket, or the Amazon S3 path contains a Region endpoint like us-east-1.amazonaws.com. For more information, see When I run an Athena query, I get an "access denied" error in the AWS Knowledge Center.
Access denied with status code: 403 error when running DDL queries on encrypted data in Amazon S3

When you may receive the error message Access Denied (Service: Amazon S3; Status Code: 403; Error Code: AccessDenied; Request ID: <request_id>) if the following conditions are true:

1. You run a DDL query like `ALTER TABLE ADD PARTITION` or `MSCK REPAIR TABLE`.
2. You have a bucket that has default encryption configured to use SSE-S3.
3. The bucket also has a bucket policy like the following that forces PutObject requests to specify the PUT headers "s3:x-amz-server-side-encryption": "true" and "s3:x-amz-server-side-encryption": "AES256".

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": "*",
      "Action": "s3:PutObject",
      "Resource": "arn:aws:s3:::<resource-name>/*",
      "Condition": {
        "Null": {
          "s3:x-amz-server-side-encryption": "true"
        }
      }
    },
    {
      "Effect": "Deny",
      "Principal": "*",
      "Action": "s3:PutObject",
      "Resource": "arn:aws:s3:::<resource-name>/*",
      "Condition": {
        "StringNotEquals": {
          "s3:x-amz-server-side-encryption": "AES256"
        }
      }
    }
  ]
}
```

In a case like this, the recommended solution is to remove the bucket policy like the one above given that the bucket's default encryption is already present.

Access denied with status code: 403 when querying an Amazon S3 bucket in another account

This error can occur when you try to query logs written by another AWS service and the second account is the bucket owner but does not own the objects in the bucket. For more information, see [I get the Amazon S3 exception “access denied with status code: 403” in Amazon Athena when I query a bucket in another account](https://aws.amazon.com/blogs/database/i-get-the-amazon-s3-exception-access-denied-with-status-code-403-in-amazon-athena-when-i-query-a-bucket-in-another-account/) in the AWS Knowledge Center or watch the Knowledge Center video.
Use IAM role credentials to connect to the Athena JDBC driver

You can retrieve a role's temporary credentials to authenticate the JDBC connection to Athena (p. 183). Temporary credentials have a maximum lifespan of 12 hours. For more information, see How can I use my IAM role credentials or switch to another IAM role when connecting to Athena using the JDBC driver? in the AWS Knowledge Center.

Query syntax issues

**FAILED: NullPointerException name is null**

If you use the AWS Glue `CreateTable` API operation or the AWS CloudFormation `AWS::Glue::Table` template to create a table for use in Athena without specifying the `TableType` property and then run a DDL query like `SHOW CREATE TABLE` or `MSCK REPAIR TABLE`, you can receive the error message **FAILED: NullPointerException Name is null.**

To resolve the error, specify a value for the `TableInput TableType` attribute as part of the AWS Glue `CreateTable` API call or `AWS CloudFormation template`. Possible values for `TableType` include EXTERNAL_TABLE or VIRTUAL_VIEW.

This requirement applies only when you create a table using the AWS Glue `CreateTable` API operation or the `AWS::Glue::Table` template. If you create a table for Athena by using a DDL statement or an AWS Glue crawler, the `TableType` property is defined for you automatically.

**Function not registered**

This error occurs when you try to use a function that Athena doesn't support. For a list of functions that Athena supports, see Functions in Amazon Athena (p. 719) or run the `SHOW FUNCTIONS` statement in the Query Editor. You can also write your own user defined function (UDF) (p. 434). For more information, see How do I resolve the "function not registered" syntax error in Athena? in the AWS Knowledge Center.

**GENERIC_INTERNAL_ERROR exceptions**

**GENERIC_INTERNAL_ERROR** exceptions can have a variety of causes, including the following:

- **GENERIC_INTERNAL_ERROR: Null** – You might see this exception under either of the following conditions:
  - You have a schema mismatch between the data type of a column in table definition and the actual data type of the dataset.
  - You are running a `CREATE TABLE AS SELECT` (CTAS) query with inaccurate syntax.

- **GENERIC_INTERNAL_ERROR: Parent builder is null** – You might see this exception when you query a table with columns of data type `array`, and you are using the OpenCSVSerDe library. The OpenCSVSerde format doesn't support the `array` data type.

- **GENERIC_INTERNAL_ERROR: Value exceeds MAX_INT** – You might see this exception when the source data column is defined with the data type `INT` and has a numeric value greater than 2,147,483,647.

- **GENERIC_INTERNAL_ERROR: Value exceeds MAX_BYTE** – You might see this exception when the source data column has a numeric value exceeding the allowable size for the data type `BYTE`. The data type `BYTE` is equivalent to `TINYINT`. `TINYINT` is an 8-bit signed integer in two's complement format with a minimum value of -128 and a maximum value of 127.
• **GENERIC_INTERNAL_ERROR: Number of partition values does not match number of filters**
  -- You might see this exception if you have inconsistent partitions on Amazon Simple Storage Service(Amazon S3) data. You might have inconsistent partitions under either of the following conditions:
  • Partitions on Amazon S3 have changed (example: new partitions were added).
  • The number of partition columns in the table do not match those in the partition metadata.

For more detailed information about each of these errors, see How do I resolve the error “GENERIC_INTERNAL_ERROR” when I query a table in Amazon Athena? in the AWS Knowledge Center.

**Number of matching groups doesn't match the number of columns**

This error occurs when you use the *Regex SerDe* (p. 337) in a CREATE TABLE statement and the number of regex matching groups doesn't match the number of columns that you specified for the table. For more information, see How do I resolve the RegexSerDe error "number of matching groups doesn't match the number of columns" in amazon Athena? in the AWS Knowledge Center.

**queryString failed to satisfy constraint: Member must have length less than or equal to 262144**

The maximum query string length in Athena (262,144 bytes) is not an adjustable quota. AWS Support can't increase the quota for you, but you can work around the issue by splitting long queries into smaller ones. For more information, see How can I increase the maximum query string length in Athena? in the AWS Knowledge Center.

**SYNTAX_ERROR: Column cannot be resolved**

This error can occur when you query a table created by an AWS Glue crawler from a UTF-8 encoded CSV file that has a byte order mark (BOM). AWS Glue doesn't recognize the BOMs and changes them to question marks, which Amazon Athena doesn't recognize. The solution is to remove the question mark in Athena or in AWS Glue.

**Throttling issues**

If your queries exceed the limits of dependent services such as Amazon S3, AWS KMS, AWS Glue, or AWS Lambda, the following messages can be expected. To resolve these issues, reduce the number of concurrent calls that originate from the same account.

<table>
<thead>
<tr>
<th>Service</th>
<th>Error message</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS Glue</td>
<td>AWSGlueException: Rate exceeded.</td>
</tr>
<tr>
<td>AWS KMS</td>
<td>You have exceeded the rate at which you may call KMS. Reduce the frequency of your calls.</td>
</tr>
<tr>
<td>AWS Lambda</td>
<td>Rate exceeded</td>
</tr>
<tr>
<td></td>
<td>TooManyRequestsException</td>
</tr>
<tr>
<td>Amazon S3</td>
<td>AmazonS3Exception: Please reduce your request rate.</td>
</tr>
</tbody>
</table>
Views

Views created in Apache Hive shell do not work in Athena

Because of their fundamentally different implementations, views created in Apache Hive shell are not compatible with Athena. To resolve this issue, re-create the views in Athena.

View is stale; it must be re-created

You can receive this error if the table that underlies a view has altered or dropped. The resolution is to recreate the view. For more information, see How can I resolve the “view is stale; it must be re-created” error in Athena? in the AWS Knowledge Center.

Workgroups

For information about troubleshooting workgroup issues, see Troubleshooting workgroups (p. 645).

Additional resources

The following pages provide additional information for troubleshooting issues with Amazon Athena.

- Athena error catalog (p. 777)
- Service Quotas (p. 785)
- Considerations and limitations for SQL queries in Amazon Athena (p. 764)
- Unsupported DDL (p. 731)
- Names for tables, databases, and columns (p. 254)
- Data types in Amazon Athena (p. 690)
- Supported SerDes and data formats (p. 303)
- Athena compression support (p. 299)
- Reserved keywords (p. 256)
- Troubleshooting workgroups (p. 645)

The following AWS resources can also be of help:

- Athena topics in the AWS knowledge center
- Amazon Athena questions on AWS re:Post
- Athena posts in the AWS big data blog

Troubleshooting often requires iterative query and discovery by an expert or from a community of helpers. If you continue to experience issues after trying the suggestions on this page, contact AWS Support (in the AWS Management Console, click Support, Support Center) or ask a question on AWS re:Post using the Amazon Athena tag.
Athena error catalog

Athena provides standardized error information to help you understand failed queries and take steps after a query failure occurs. The AthenaError feature includes an ErrorCategory field and an ErrorType field. ErrorCategory specifies whether the cause of the failed query is due to system error, user error, or other error. ErrorType provides more granular information regarding the source of the failure. By combining the two fields, you can get a better understanding of the circumstances surrounding and causes for the specific error that occurred.

Error category

The following table lists the Athena error category values and their meanings.

<table>
<thead>
<tr>
<th>Error category</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>2</td>
<td>USER</td>
</tr>
<tr>
<td>3</td>
<td>OTHER</td>
</tr>
</tbody>
</table>

Error type reference

The following table lists the Athena error type values and their meanings.

<table>
<thead>
<tr>
<th>Error type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Query exhausted resources at this scale factor</td>
</tr>
<tr>
<td>1</td>
<td>Query exhausted resources at this scale factor</td>
</tr>
<tr>
<td>2</td>
<td>Query exhausted resources at this scale factor</td>
</tr>
<tr>
<td>3</td>
<td>Query exhausted resources at this scale factor</td>
</tr>
<tr>
<td>4</td>
<td>Query exhausted resources at this scale factor</td>
</tr>
<tr>
<td>5</td>
<td>Query exhausted resources at this scale factor</td>
</tr>
<tr>
<td>6</td>
<td>Query exhausted resources at this scale factor</td>
</tr>
<tr>
<td>7</td>
<td>Query exhausted resources at this scale factor</td>
</tr>
<tr>
<td>8</td>
<td>Query exhausted resources at this scale factor</td>
</tr>
<tr>
<td>100</td>
<td>Internal service error</td>
</tr>
<tr>
<td>200</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>201</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>202</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>203</td>
<td>Driver error</td>
</tr>
<tr>
<td>204</td>
<td>The metastore had an error</td>
</tr>
<tr>
<td>Error type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>205</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>206</td>
<td>Query timed out</td>
</tr>
<tr>
<td>207</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>208</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>209</td>
<td>Failed to cancel query</td>
</tr>
<tr>
<td>210</td>
<td>Query timed out</td>
</tr>
<tr>
<td>211</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>212</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>213</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>214</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>215</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>216</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>217</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>218</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>219</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>220</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>221</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>222</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>223</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>224</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>225</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>226</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>227</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>228</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>229</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>230</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>231</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>232</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>233</td>
<td>Iceberg error</td>
</tr>
<tr>
<td>234</td>
<td>Lake Formation error</td>
</tr>
<tr>
<td>235</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>Error type</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>236</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>237</td>
<td>Serialization error</td>
</tr>
<tr>
<td>238</td>
<td>Failed to upload metadata to Amazon S3</td>
</tr>
<tr>
<td>239</td>
<td>General persistence error</td>
</tr>
<tr>
<td>240</td>
<td>Failed to submit query</td>
</tr>
<tr>
<td>300</td>
<td>Internal service error</td>
</tr>
<tr>
<td>301</td>
<td>Internal service error</td>
</tr>
<tr>
<td>302</td>
<td>Internal service error</td>
</tr>
<tr>
<td>303</td>
<td>Internal service error</td>
</tr>
<tr>
<td>400</td>
<td>Internal service error</td>
</tr>
<tr>
<td>401</td>
<td>Failed to write query results to Amazon S3</td>
</tr>
<tr>
<td>402</td>
<td>Failed to write query results to Amazon S3</td>
</tr>
<tr>
<td>1000</td>
<td>User error</td>
</tr>
<tr>
<td>1001</td>
<td>Data error</td>
</tr>
<tr>
<td>1002</td>
<td>Data error</td>
</tr>
<tr>
<td>1003</td>
<td>DDL task failed</td>
</tr>
<tr>
<td>1004</td>
<td>Schema error</td>
</tr>
<tr>
<td>1005</td>
<td>Serialization error</td>
</tr>
<tr>
<td>1006</td>
<td>Syntax error</td>
</tr>
<tr>
<td>1007</td>
<td>Data error</td>
</tr>
<tr>
<td>1008</td>
<td>Query rejected</td>
</tr>
<tr>
<td>1009</td>
<td>Query failed</td>
</tr>
<tr>
<td>1010</td>
<td>Internal service error</td>
</tr>
<tr>
<td>1011</td>
<td>Query canceled by user</td>
</tr>
<tr>
<td>1012</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>1013</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>1014</td>
<td>Query canceled by user</td>
</tr>
<tr>
<td>1100</td>
<td>Invalid argument provided</td>
</tr>
<tr>
<td>1101</td>
<td>Invalid property provided</td>
</tr>
<tr>
<td>1102</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>1103</td>
<td>Invalid table property provided</td>
</tr>
<tr>
<td>Error type</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>1104</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>1105</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>1106</td>
<td>Invalid function argument provided</td>
</tr>
<tr>
<td>1107</td>
<td>Invalid view</td>
</tr>
<tr>
<td>1108</td>
<td>Failed to register function</td>
</tr>
<tr>
<td>1109</td>
<td>Provided Amazon S3 path not found</td>
</tr>
<tr>
<td>1110</td>
<td>Provided table or view does not exist</td>
</tr>
<tr>
<td>1200</td>
<td>Query not supported</td>
</tr>
<tr>
<td>1201</td>
<td>Provided decoder not supported</td>
</tr>
<tr>
<td>1202</td>
<td>Query type not supported</td>
</tr>
<tr>
<td>1300</td>
<td>General not found error</td>
</tr>
<tr>
<td>1301</td>
<td>General entity not found</td>
</tr>
<tr>
<td>1302</td>
<td>File not found</td>
</tr>
<tr>
<td>1303</td>
<td>Provided function or function implementation not found</td>
</tr>
<tr>
<td>1304</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>1305</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>1306</td>
<td>Amazon S3 bucket not found</td>
</tr>
<tr>
<td>1307</td>
<td>Selected engine not found</td>
</tr>
<tr>
<td>1308</td>
<td>Query engine had an internal error</td>
</tr>
<tr>
<td>1400</td>
<td>Throttling error</td>
</tr>
<tr>
<td>1401</td>
<td>Query failed due to AWS Glue throttling</td>
</tr>
<tr>
<td>1402</td>
<td>Query failed due to too many table versions in AWS Glue</td>
</tr>
<tr>
<td>1403</td>
<td>Query failed due to Amazon S3 throttling</td>
</tr>
<tr>
<td>1404</td>
<td>Query failed due to Amazon Athena throttling</td>
</tr>
<tr>
<td>1405</td>
<td>Query failed due to Amazon Athena throttling</td>
</tr>
<tr>
<td>1406</td>
<td>Query failed due to Amazon Athena throttling</td>
</tr>
<tr>
<td>1500</td>
<td>Permission error</td>
</tr>
<tr>
<td>1501</td>
<td>Amazon S3 permission error</td>
</tr>
<tr>
<td>99999</td>
<td>Internal service error</td>
</tr>
</tbody>
</table>
Performance tuning in Athena

This topic provides general information and specific suggestions for improving the performance of Athena when you have large amounts of data and experience memory usage or performance issues.

Physical limits

By default, Athena limits (p. 785) the runtime of DML queries to 30 minutes and DDL queries to 600 minutes. Queries that run beyond these limits are automatically cancelled without charge. If a query runs out of memory or a node crashes during processing, errors like the following can occur:

- INTERNAL_ERROR_QUERY_ENGINE
- EXCEEDED_MEMORY_LIMIT: Query exceeded local memory limit
- Query exhausted resources at this scale factor
- Encountered too many errors talking to a worker node. The node may have crashed or be under too much load.

Query optimization techniques

For queries that require resources beyond existing limits, you can either optimize the query or restructure the data being queried. To optimize your queries, consider the suggestions in this section.

- Data size (p. 781)
- File formats (p. 782)
- Joins, grouping, and unions (p. 782)
- Partitioning (p. 783)
- Window functions (p. 783)
- Use more efficient functions (p. 783)

Data size

Avoid single large files – If your file size is extremely large, try to break up the file into smaller files and use partitions to organize them.

Read a smaller amount of data at once – Scanning a large amount of data at one time can slow down the query and increase cost. Use partitions or filters to limit the files to be scanned.

Avoid having too many columns – The message GENERIC_INTERNAL_ERROR: io.airlift.bytecode.CompilationException can occur when Athena fails to compile the query to bytecode. This exception is usually caused by having too many columns in the query. Reduce the number of the columns in the query or create subqueries and use a JOIN that retrieves a smaller amount of data.
Avoid large query outputs – A large amount of output data can slow performance. To work around this, try using CTAS (p. 744) to create a new table with the result of the query or INSERT INTO (p. 697) to append new results into an existing table.

Avoid CTAS queries with a large output – CTAS queries can also use a large amount of memory. If you are outputting large amount of data, try separating the task into smaller queries.

If possible, avoid having a large number of small files – Amazon S3 has a limit of 5500 requests per second. Athena queries share the same limit. If you need to scan millions of small objects in a single query, your query can be easily throttled by Amazon S3. To avoid excessive scanning, use AWS Glue ETL to periodically compact your files or partition the table and add partition key filters. For more information, see Reading input files in larger groups in the AWS Glue Developer Guide or How can I configure an AWS Glue ETL job to output larger files? in the AWS Knowledge Center.

Avoid scanning an entire table – Use the following techniques to avoid scanning entire tables:

- Limit the use of "**". Try not to select all columns unless necessary.
- Avoid scanning the same table multiple times in the same query
- Use filters to reduce the amount of data to be scanned.
- Whenever possible, add a LIMIT clause.

Avoid referring to many views and tables in a single query – Because queries with many views and/or tables must load a large amount of data, out of memory errors can occur. If possible, avoid referring to an excessive number of views or tables in a single query.

Avoid large JSON strings – If data is stored in a single JSON string and the size of the JSON data is large, out of memory errors can occur when the JSON data is processed.

File formats

Use an efficient file format such as parquet or ORC – To dramatically reduce query running time and costs, use compressed Parquet or ORC files to store your data. To convert your existing dataset to those formats in Athena, you can use CTAS. For more information, see Using CTAS and INSERT INTO for ETL and data analysis (p. 290).

Switch between ORC and parquet formats – Experience shows that the same set of data can have significant differences in processing time depending on whether it is stored in ORC or Parquet format. If you are experiencing performance issues, try a different format.

Hudi queries – Because Hudi queries (p. 420) bypass the native reader and split generator for files in parquet format, they can be slow. Keep this in mind when querying Hudi datasets.

Joins, grouping, and unions

Reduce the usage of memory intensive operations – Operations like JOIN, GROUP BY, ORDER BY, and UNION all require loading large amount of data into memory. To speed up your query, find other ways to achieve the same results, or add a clause like LIMIT to the outer query whenever possible.

Consider using UNION ALL – To eliminate duplicates, UNION builds a hash table, which consumes memory. If your query does not require the elimination of duplicates, consider using UNION ALL for better performance.

Use CTAS as an intermediary step to speed up JOIN operations – Instead of loading and processing intermediary data with every query, use CTAS to persist the intermediary data into Amazon S3. This can help speed up the performance of operations like JOIN.
Partitioning

**Limit the number of partitions in a table** – When a table has more than 100,000 partitions, queries can be slow because of the large number of requests sent to AWS Glue to retrieve partition information. To resolve this issue, try one of the following options:

- Use `ALTER TABLE DROP PARTITION (p. 735)` to remove stale partitions.
- If your partition pattern is predictable, use `partition projection (p. 265)`.

**Remove old partitions even if they are empty** – Even if a partition is empty, the metadata of the partition is still stored in AWS Glue. Loading these unneeded partitions can increase query runtimes. To remove the unneeded partitions, use `ALTER TABLE DROP PARTITION (p. 735)`.

**Look up a single partition** – When looking up a single partition, try to provide all partition values so that Athena can locate the partition with a single call to AWS Glue. Otherwise, Athena must retrieve all partitions and filter them. This can be costly and greatly increase the planning time for your query. If you have a predictable partition pattern, you can use `partition projection (p. 265)` to avoid the partition look up calls to AWS Glue.

**Set reasonable partition projection properties** – When using `partition projection (p. 265)`, Athena tries to create a partition object for every partition name. Because of this, make sure that the table properties that you define do not create a near infinite amount of possible partitions.

**To add new partitions frequently, use `ALTER TABLE ADD PARTITION`** – If you use `MSCK REPAIR TABLE` to add new partitions frequently (for example, on a daily basis) and are experiencing query timeouts, consider using `ALTER TABLE ADD PARTITION (p. 733)`. `MSCK REPAIR TABLE` is best used when creating a table for the first time or when there is uncertainty about parity between data and partition metadata.

**Avoid using coalesce() in a WHERE clause with partitioned columns** – Under some circumstances, using the `coalesce()` or other functions in a `WHERE` clause against partitioned columns might result in reduced performance. If this occurs, try rewriting your query to provide the same functionality without using `coalesce()`.

Window functions

**Minimize the use of window functions** – Window functions such as `rank()` are memory intensive. In general, window functions require an entire dataset to be loaded into a single Athena node for processing. With an extremely large dataset, this can risk crashing the node. To avoid this, try the following options:

- Filter the data and run window functions on a subset of the data.
- Use the `PARTITION BY` clause with the window function whenever possible.
- Find an alternative way to construct the query.

**Use more efficient functions**

Replace `row_number() OVER (...) as rnk ... WHERE rnk = 1` – To speed up a query with a `row_number()` clause like this, replace this syntax with a combination of `GROUP BY`, `ORDER BY`, and `LIMIT 1`.

Use regular expressions instead of LIKE on large strings – Queries that include clauses such as LIKE `"%string%"` on large strings can be very costly. Consider using the `regexp_like()` function and a regular expression instead.
Use `max()` instead of `element_at(array_sort(), 1)` – For increased speed, replace the nested functions `element_at(array_sort(), 1)` with `max()`.

**Additional resources**

For additional information about performance tuning in Athena, consider the following resources:

- Read the AWS Big Data blog post [Top 10 performance tuning tips for Amazon Athena](https://aws.amazon.com/blogs/big-data/top-10-performance-tuning-tips-for-amazon-athena)
- Read other Athena posts in the AWS big data blog
- Ask a question on [AWS re:Post](https://aws.amazon.com/repost/) using the [Amazon Athena](https://aws.amazon.com/athena/) tag
- Consult the [Athena topics in the AWS knowledge center](https://aws.amazon.com/knowledge-center/)
- Contact AWS Support (in the AWS Management Console, click [Support, Support Center](https://aws.amazon.com/support/))
Service Quotas

**Note**
The Service Quotas console provides information about Amazon Athena quotas. You can also use the Service Quotas console to request quota increases for the quotas that are adjustable. For general information about AWS service quotas, see AWS service quotas in the AWS General Reference.

Queries

Your account has the following query-related quotas for Amazon Athena. For details, see the Amazon Athena endpoints and quotas page of the AWS General Reference.

- **Active DDL queries** – The number of active DDL queries. DDL queries include CREATE TABLE and ALTER TABLE ADD PARTITION queries.
- **DDL query timeout** – The maximum amount of time in minutes a DDL query can run before it gets cancelled.
- **Active DML queries** – The number of active DML queries. DML queries include SELECT, CREATE TABLE AS (CTAS), and INSERT INTO queries. The specific quotas vary by AWS Region.
- **DML query timeout** – The maximum amount of time in minutes a DML query can run before it gets cancelled.

These are soft quotas; you can use the Athena Service Quotas console to request a quota increase.

Athena processes queries by assigning resources based on the overall service load and the number of incoming requests. Your queries may be temporarily queued before they run. Asynchronous processes pick up the queries from queues and run them on physical resources as soon as the resources become available and for as long as your account configuration permits.

A DML or DDL query quota includes both running and queued queries. For example, if your DML query quota is 25 and your total of running and queued queries is 26, query 26 will result in a TooManyRequestsException error.

**Query string length**

The maximum allowed query string length is 262144 bytes, where the strings are encoded in UTF-8. This is not an adjustable quota. However, you can work around this limitation by splitting long queries into multiple smaller queries. For more information, see How can I increase the maximum query string length in Athena? in the AWS Knowledge Center.

**Note**
If you require a greater query string length, provide feedback at athena-feedback@amazon.com with the details of your use case, or contact AWS Support.

Workgroups

When you work with Athena workgroups, remember the following points:
• Athena service quotas are shared across all workgroups in an account.
• The maximum number of workgroups you can create per Region in an account is 1000.
• The maximum number of prepared statements in a workgroup is 1000.
• The maximum number of tags per workgroup is 50. For more information, see Tag restrictions (p. 658).

Databases, tables, and partitions

• If you are using the AWS Glue Data Catalog with Athena, see AWS Glue endpoints and quotas for service quotas on tables, databases, and partitions – for example, the maximum number of databases or tables per account.
• Although Athena supports querying AWS Glue tables that have 10 million partitions, Athena cannot read more than 1 million partitions in a single scan.
• If you are not using AWS Glue Data Catalog, the number of partitions per table is 20,000. You can request a quota increase.

Note
If you have not yet migrated to AWS Glue Data Catalog, see Upgrading to the AWS Glue Data Catalog Step-by-Step (p. 31) for migration instructions.

Amazon S3 buckets

When you work with Amazon S3 buckets, remember the following points:

• Amazon S3 has a default service quota of 100 buckets per account.
• Athena requires a separate bucket to log results.
• You can request a quota increase of up to 1,000 Amazon S3 buckets per AWS account.

Per account API call quotas

Athena APIs have the following default quotas for the number of calls to the API per account (not per query):

<table>
<thead>
<tr>
<th>API name</th>
<th>Default number of calls per second</th>
<th>Burst capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>BatchGetNamedQuery,</td>
<td>5</td>
<td>up to 10</td>
</tr>
<tr>
<td>ListNamedQueries,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ListQueryExecutions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CreateNamedQuery,</td>
<td>5</td>
<td>up to 20</td>
</tr>
<tr>
<td>DeleteNamedQuery,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetNamedQuery</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BatchGetQueryExecution</td>
<td>20</td>
<td>up to 40</td>
</tr>
<tr>
<td>StartQueryExecution,</td>
<td>20</td>
<td>up to 80</td>
</tr>
<tr>
<td>StopQueryExecution</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetQueryExecution,</td>
<td>100</td>
<td>up to 200</td>
</tr>
<tr>
<td>GetQueryResults</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
For example, you can make up to 20 calls per second for `StartQueryExecution`. In addition, if this API is not called for 4 seconds, your account accumulates a burst capacity of up to 80 calls. In this case, your application can make up to 80 calls to this API in burst mode.

If you use any of these APIs and exceed the default quota for the number of calls per second, or the burst capacity in your account, the Athena API issues an error similar to the following: "ClientError: An error occurred (ThrottlingException) when calling the `<API_name>` operation: Rate exceeded." Reduce the number of calls per second, or the burst capacity for the API for this account. To request a quota increase, contact AWS Support. Open the AWS Support center page, sign in if necessary, and choose Create case. Choose Service limit increase. Complete and submit the form.

**Note**

This quota cannot be changed in the Athena Service Quotas console.


**Code samples**

Use the examples in this topic as a starting point for writing Athena applications using the SDK for Java 2.x. For more information about running the Java code examples, see the Amazon Athena Java readme on the AWS code examples repository on GitHub.

- **Java Code Examples**
  - Constants (p. 788)
  - Create a client to access Athena (p. 788)
- **Working with Query Executions**
  - Start query execution (p. 789)
  - Stop query execution (p. 791)
  - List query executions (p. 793)
- **Working with Named Queries**
  - Create a named query (p. 794)
  - Delete a named query (p. 795)
  - List query executions (p. 793)

**Note**

These samples use constants (for example, ATHENA_SAMPLE_QUERY) for strings, which are defined in an ExampleConstants.java class declaration. Replace these constants with your own strings or defined constants.

**Constants**

The ExampleConstants.java class demonstrates how to query a table created by the Getting started (p. 9) tutorial in Athena.

```java
package aws.example.athena;

class ExampleConstants {

  public static final int CLIENT_EXECUTION_TIMEOUT = 100000;
  public static final String ATHENA_OUTPUT_BUCKET = "s3://bucketscott2"; // change the Amazon S3 bucket name to match your environment
  //  Demonstrates how to query a table with a comma-separated value (CSV) table.  For information, see
  public static final String ATHENA_SAMPLE_QUERY = "SELECT * FROM scott2;"; // change the Query statement to match your environment
  public static final long SLEEP_AMOUNT_IN_MS = 1000;
  public static final String ATHENA_DEFAULT_DATABASE = "mydatabase"; // change the database to match your database
}
```

Create a client to access Athena

The AthenaClientFactory.java class shows how to create and configure an Amazon Athena client.
The StartQueryExample shows how to submit a query to Athena, wait until the results become available, and then process the results.
String queryExecutionId = submitAthenaQuery(athenaClient);
waitForQueryToComplete(athenaClient, queryExecutionId);
processResultRows(athenaClient, queryExecutionId);
athenaClient.close();

// Submits a sample query to Amazon Athena and returns the execution ID of the query. 
public static String submitAthenaQuery(AthenaClient athenaClient) { 
    try {
        // The QueryExecutionContext allows us to set the database.
        QueryExecutionContext queryExecutionContext = QueryExecutionContext.builder()
            .database(ExampleConstants.ATHENA_DEFAULT_DATABASE)
            .build();

        // The result configuration specifies where the results of the query should go.
        ResultConfiguration resultConfiguration = ResultConfiguration.builder()
            .outputLocation(ExampleConstants.ATHENA_OUTPUT_BUCKET)
            .build();

        StartQueryExecutionRequest startQueryExecutionRequest =
            StartQueryExecutionRequest.builder()
            .queryString(ExampleConstants.ATHENA_SAMPLE_QUERY)
            .queryExecutionContext(queryExecutionContext)
            .resultConfiguration(resultConfiguration)
            .build();

        StartQueryExecutionResponse startQueryExecutionResponse =
            athenaClient.startQueryExecution(startQueryExecutionRequest);
        return startQueryExecutionResponse.queryExecutionId();
    } catch (AthenaException e) {
        e.printStackTrace();
        System.exit(1);
    }
    return "";
}

// Wait for an Amazon Athena query to complete, fail or to be cancelled.
public static void waitForQueryToComplete(AthenaClient athenaClient, String queryExecutionId) throws InterruptedException {
    GetQueryExecutionRequest getQueryExecutionRequest =
        GetQueryExecutionRequest.builder()
        .queryExecutionId(queryExecutionId)
        .build();

    GetQueryExecutionResponse getQueryExecutionResponse;
    boolean isQueryStillRunning = true;
    while (isQueryStillRunning) {
        getQueryExecutionResponse =
            athenaClient.getQueryExecution(getQueryExecutionRequest);
        String queryState =
            getQueryExecutionResponse.queryExecution().status().state().toString();
        if (queryState.equals(QueryExecutionState.FAILED.toString())) {
            throw new RuntimeException("The Amazon Athena query failed to run with error message: " + getQueryExecutionResponse.queryExecution().status().stateChangeReason());
        } else if (queryState.equals(QueryExecutionState.CANCELLED.toString())) {
            throw new RuntimeException("The Amazon Athena query was cancelled.");
        } else if (queryState.equals(QueryExecutionState.SUCCEEDED.toString())) {
            isQueryStillRunning = false;
        } else {
            // Sleep an amount of time before retrying again.
            Thread.sleep(ExampleConstants.SLEEP_AMOUNT_IN_MS);
        }
        System.out.println("The current status is: " + queryState);
    }
}
// This code retrieves the results of a query
public static void processResultRows(AthenaClient athenaClient, String queryExecutionId) { try {
    // Max Results can be set but if its not set,
    // it will choose the maximum page size.
    GetQueryResultsRequest getQueryResultsRequest = GetQueryResultsRequest.builder()
        .queryExecutionId(queryExecutionId)
        .build();

    GetQueryResultsIterable getQueryResultsResults = athenaClient.getQueryResultsPaginator(getQueryResultsRequest);
    for (GetQueryResultsResponse result : getQueryResultsResults) {
        List<ColumnInfo> columnInfoList = result.resultSet().resultSetMetadata().columnInfo();
        List<Row> results = result.resultSet().rows();
        processRow(results, columnInfoList);
    }
} catch (AthenaException e) {
    e.printStackTrace();
    System.exit(1);
}

private static void processRow(List<Row> row, List<ColumnInfo> columnInfoList) {
    for (Row myRow : row) {
        List<Datum> allData = myRow.data();
        for (Datum data : allData) {
            System.out.println("The value of the column is " + data.varCharValue());
        }
    }
}

Stop query execution

The StopQueryExecutionExample runs an example query, immediately stops the query, and checks the status of the query to ensure that it was canceled.
**Before running this Java V2 code example, set up your development environment, including your credentials.**

* For more information, see the following documentation topic:
* https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html

```java
public class StopQueryExecutionExample {
    public static void main(String[] args) {
        AthenaClient athenaClient = AthenaClient.builder()
            .region(Region.US_WEST_2)
            .credentialsProvider(ProfileCredentialsProvider.create())
            .build();

        String sampleQueryExecutionId = submitAthenaQuery(athenaClient);
        stopAthenaQuery(athenaClient, sampleQueryExecutionId);
        athenaClient.close();
    }

    public static void stopAthenaQuery(AthenaClient athenaClient, String sampleQueryExecutionId) {
        try {
            StopQueryExecutionRequest stopQueryExecutionRequest = StopQueryExecutionRequest.builder()
                .queryExecutionId(sampleQueryExecutionId)
                .build();

            athenaClient.stopQueryExecution(stopQueryExecutionRequest);

            // Ensure that the query was stopped.
            GetQueryExecutionRequest getQueryExecutionRequest = GetQueryExecutionRequest.builder()
                .queryExecutionId(sampleQueryExecutionId)
                .build();

            GetQueryExecutionResponse getQueryExecutionResponse = athenaClient.getQueryExecution(getQueryExecutionRequest);
            if (getQueryExecutionResponse.queryExecution().status().state().equals(QueryExecutionState.CANCELLED)) {
                System.out.println("The Amazon Athena query has been cancelled!");
            }
        } catch (AthenaException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    // Submits an example query and returns a query execution ID value
    public static String submitAthenaQuery(AthenaClient athenaClient) {
        try {
            QueryExecutionContext queryExecutionContext = QueryExecutionContext.builder()
                .database(ExampleConstants.ATHENA_DEFAULT_DATABASE)
                .build();

            ResultConfiguration resultConfiguration = ResultConfiguration.builder()
                .outputLocation(ExampleConstants.ATHENA_OUTPUT_BUCKET)
                .build();

            StartQueryExecutionRequest startQueryExecutionRequest = StartQueryExecutionRequest.builder()
                .queryExecutionContext(queryExecutionContext)
                .resultConfiguration(resultConfiguration)
                .build();

            return athenaClient.startQueryExecution(startQueryExecutionRequest).
```

---

792
The `ListQueryExecutionsExample` shows how to obtain a list of query execution IDs.

```java
package aws.example.athena;

import software.amazon.awssdk.auth.credentials.ProfileCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.athena.AthenaClient;
import software.amazon.awssdk.services.athena.model.AthenaException;
import software.amazon.awssdk.services.athena.model.ListQueryExecutionsRequest;
import software.amazon.awssdk.services.athena.model.ListQueryExecutionsResponse;
import software.amazon.awssdk.services.athena.paginators.ListQueryExecutionsIterable;
import java.util.List;

/**
 * Before running this Java V2 code example, set up your development environment, including your credentials.
 * For more information, see the following documentation topic:
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 */
public class ListQueryExecutionsExample {
    public static void main(String[] args) {
        AthenaClient athenaClient = AthenaClient.builder()
            .region(Region.US_WEST_2)
            .credentialsProvider(ProfileCredentialsProvider.create())
            .build();

        listQueryIds(athenaClient);
        athenaClient.close();
    }

    public static void listQueryIds(AthenaClient athenaClient) {
        try {
            ListQueryExecutionsRequest listQueryExecutionsRequest = ListQueryExecutionsRequest.builder().build();
            ListQueryExecutionsIterable listQueryExecutionResponses = athenaClient.listQueryExecutionsPaginator(listQueryExecutionsRequest);
        } catch (AthenaException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```
Create a named query

The CreateNamedQueryExample shows how to create a named query.

```java
package aws.example.athena;
import software.amazon.awssdk.auth.credentials.ProfileCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.athena.AthenaClient;
import software.amazon.awssdk.services.athena.model.AthenaException;
import software.amazon.awssdk.services.athena.model.CreateNamedQueryRequest;

public class CreateNamedQueryExample {
    public static void main(String[] args) {
        final String USAGE = "\n" +
            "Usage:
" +
            "    <name>
" +
            "Where:
" +
            "    name - the name of the Amazon Athena query. \n\n";

        if (args.length != 1) {
            System.out.println(USAGE);
            System.exit(1);
        }

        String name = args[0];
        AthenaClient athenaClient = AthenaClient.builder()
            .region(Region.US_WEST_2)
            .credentialsProvider(ProfileCredentialsProvider.create())
            .build();

        createNamedQuery(athenaClient, name);
        athenaClient.close();
    }

    public static void createNamedQuery(AthenaClient athenaClient, String name) {
        try {
            // Create the named query request.
        }
    }
}
```
CreateNamedQueryRequest createNamedQueryRequest =
CreateNamedQueryRequest.builder()
    .database(ExampleConstants.ATHENA_DEFAULT_DATABASE)
    .queryString(ExampleConstants.ATHENA_SAMPLE_QUERY)
    .description("Sample Description")
    .name(name)
    .build();

athenaClient.createNamedQuery(createNamedQueryRequest);
System.out.println("Done");

} catch (AthenaException e) {
    e.printStackTrace();
    System.exit(1);
}

}
List named queries

The ListNamedQueryExample shows how to obtain a list of named query IDs.

```java
package aws.example.athena;

import software.amazon.awssdk.auth.credentials.ProfileCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.athena.AthenaClient;
import software.amazon.awssdk.services.athena.model.AthenaException;
import software.amazon.awssdk.services.athena.model.ListNamedQueriesRequest;
import software.amazon.awssdk.services.athena.model.ListNamedQueriesResponse;
import software.amazon.awssdk.services.athena.paginators.ListNamedQueriesIterable;
import java.util.List;

/**
 * Before running this Java V2 code example, set up your development environment, including your credentials.
 */
```
public class ListNamedQueryExample {
    public static void main(String[] args) {
        AthenaClient athenaClient = AthenaClient.builder()
            .region(Region.US_WEST_2)
            .credentialsProvider(ProfileCredentialsProvider.create())
            .build();
        listNamedQueries(athenaClient);
        athenaClient.close();
    }

    public static void listNamedQueries(AthenaClient athenaClient) {
        try{
            ListNamedQueriesRequest listNamedQueriesRequest = ListNamedQueriesRequest.builder()
                .build();
            ListNamedQueriesIterable listNamedQueriesResponses = athenaClient.listNamedQueriesPaginator(listNamedQueriesRequest);
            for (ListNamedQueriesResponse listNamedQueriesResponse : listNamedQueriesResponses) {
                List<String> namedQueryIds = listNamedQueriesResponse.namedQueryIds();
                System.out.println(namedQueryIds);
            }
        } catch (AthenaException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
Release notes

Describes Amazon Athena features, improvements, and bug fixes by release date.

Topics

• Athena release notes for 2022 (p. 798)
• Athena release notes for 2021 (p. 808)
• Athena release notes for 2020 (p. 818)
• Athena release notes for 2019 (p. 822)
• Athena release notes for 2018 (p. 829)
• Athena release notes for 2017 (p. 834)

Athena release notes for 2022

November 18, 2022

Published on 2022-11-18

You can now use the Amazon Athena connector for IBM Db2 to query Db2 from Athena. For example, you can run analytical queries over a data warehouse on Db2 and a data lake in Amazon S3.

The Amazon Athena Db2 connector exposes several configuration options through Lambda environment variables. For information about configuration options, parameters, connection strings, deployment, and limitations, see Amazon Athena IBM Db2 connector (p. 93).

November 17, 2022

Published on 2022-11-17

Apache Iceberg support in Athena engine version 3 now offers the following enhanced ACID transaction features:

• **ORC and Avro support** – Create Iceberg tables using the Apache Avro and Apache ORC row and column-based file formats. Support for these formats is in addition to the existing support for Parquet.

• **MERGE INTO** – Use the MERGE INTO command to merge data at scale efficiently. MERGE INTO combines the INSERT, UPDATE, and DELETE operations into one transaction. This reduces the processing overhead in your data pipeline and takes less SQL to write. For more information, see Updating Iceberg table data (p. 513) and MERGE INTO (p. 701).

• **CTAS and VIEW support** – Use the CREATE TABLE AS SELECT (CTAS) and CREATE VIEW statements with Iceberg tables. For more information, see CREATE TABLE AS (p. 744) and CREATE VIEW (p. 750).

• **VACUUM support** – You can use the VACUUM statement to optimize your data lake by deleting snapshots and data that are no longer required. You can use this feature to improve read performance and meet regulatory requirements like GDPR. For more information, see Optimizing Iceberg tables (p. 515) and VACUUM (p. 702).

These new features require Athena engine version 3 and are available in all Regions where Athena is supported. You can use them with the Athena console, drivers (p. 183), or API.
November 14, 2022

Published on 2022-11-14

Amazon Athena now supports IPv6 endpoints for inbound connections that you can use to invoke Athena functions over IPv6. You can use this feature to meet IPv6 compliance requirements. It also removes the need for additional networking equipment to handle address translation between IPv4 and IPv6.

To use this feature, configure your applications to use the new Athena dual-stack endpoints, which support both IPv4 and IPv6. Dual-stack endpoints use the format athena.region.api.aws. For example, the dual-stack endpoint in the US East (N. Virginia) Region is athena.us-east-1.api.aws.

When you make a request to a dual-stack Athena endpoint, the endpoint resolves to an IPv6 or an IPv4 address depending on the protocol used by your network and client. To connect programmatically to an AWS service, you can use the AWS CLI or AWS SDK to specify an endpoint.

For more information on service endpoints, see AWS service endpoints. To learn more about Athena's service endpoints, see Amazon Athena endpoints and quotas in the AWS documentation.

You can use the new Athena dual-stack endpoints for inbound connections at no additional cost. Dual-stack endpoints are generally available in all AWS Regions except the China (Beijing), China (Ningxia), and AWS GovCloud (US) Regions.

November 11, 2022

Published on 2022-11-11

Athena announces the following fixes and improvements.

- **Expanded Lake Formation fine-grained access control** – You can now use AWS Lake Formation fine-grained access control policies in Athena queries for data stored in any supported file or table format. You can use fine-grained access control in Lake Formation to restrict access to data in query results using data filters to achieve column-level, row-level, and cell-level security. Supported table formats in Athena include Apache Iceberg, Apache Hudi, and Apache Hive. Expanded fine-grained access control is available in all regions supported by Athena. The expanded table and file format support requires Athena engine version 3 (p. 672), which offers new features and improved query performance, but does not change how you set up fine-grained access control policies in Lake Formation.

Use of this expanded fine-grained access control in Athena has the following considerations:

- **EXPLAIN** – Row or cell filtering information defined in Lake Formation and query statistics information are not shown in the output of EXPLAIN and EXPLAIN ANALYZE. For information about EXPLAIN in Athena, see Using EXPLAIN and EXPLAIN ANALYZE in Athena (p. 703).

- **External Hive metastores** – Apache Hive hidden columns cannot be used for fine-grained access control filtering, and Apache Hive hidden system tables are not supported by fine-grained access control. For more information, see Considerations and limitations (p. 37) in the topic Using Athena Data Connector for External Hive Metastore (p. 35).

- **Query statistics** – Stage-level input and output row count and data size information are not shown in Athena query statistics when a query has row-level filters defined in Lake Formation. For information about seeing statistics for Athena queries, see Viewing statistics and execution details for completed queries (p. 357) and GetQueryRuntimeStatistics.

- **Workgroups** – Users in the same Athena workgroup can see the data that Lake Formation fine-grained access control has configured to be accessible to the workgroup. For information about using Athena to query data registered with Lake Formation, see Using Athena to query data registered with AWS Lake Formation (p. 580).
For information about using fine-grained access control in Lake Formation, see Manage fine-grained access control using AWS Lake Formation in the AWS Big Data Blog.

- **Athena Federated Query** – Athena Federated Query now preserves the original casing of field names in struct objects. Previously, struct field names were automatically made lower case.

### November 8, 2022

Published on 2022-11-08

You can now use the query result reuse caching feature to accelerate repeat queries in Athena. A repeat query is a SQL query identical to one submitted just recently that produces the same results. When you need to run identical multiple queries, result reuse caching can decrease the time required to produce results. Result reuse caching also lowers costs by reducing the number of bytes scanned.

For more information, see Reusing query results (p. 354).

### October 13, 2022

Published on 2022-10-13

Athena announces Athena engine version 3.

Athena has upgraded its SQL query engine to include the latest features from the Trino open source project. In addition to supporting all the features of Athena engine version 2, Athena engine version 3 includes over 50 new SQL functions, 30 new features, and more than 90 query performance improvements. With today's launch, Athena is also introducing a continuous integration approach to open source software management that improves currency with the Trino and Presto projects so that you get faster access to community improvements, integrated and tuned within the Athena engine.

For more information, see Athena engine version 3 (p. 672).

### October 10, 2022

Published on 2022-10-10

Athena releases JDBC driver version 2.0.33. The JDBC 2.0.33 driver includes the following changes:

- New driver version, JDBC version, and plugin name properties were added to the user-agent string in the credentials provider class.
- Error messages were corrected and necessary information added.
- Prepared statements are now deallocated if the connection is closed or the Athena prepared statement execution fails.

For more information, and to download the new drivers, release notes, and documentation, see Using Athena with the JDBC driver (p. 183).

### September 23, 2022

Published on 2022-09-26

The Amazon Athena Neptune connector now supports case insensitive matching on column and table names.
• The Neptune data source connector can resolve column names on Neptune tables that use casing even if the column names are all lower cased in the table in AWS Glue. To enable this behavior, set the enable_caseinsensitivematch environment variable to true on the Neptune connector Lambda function.

• Because AWS Glue supports only lower case table names, when you create a AWS Glue table for Neptune, specify the AWS Glue table parameter "glabel" = table_name.

For more information about the Neptune connector, see Amazon Athena Neptune connector (p. 117).

September 13, 2022

Published on 2022-09-13

Athena announces the following fixes and improvements.

• **External Hive metastore** – Athena now returns NULL instead of throwing an exception when a WHERE clause includes a partition that doesn't exist in an external Hive metastore (p. 35) (EHMS). The new behavior matches that of the AWS Glue Data Catalog.

• **Parameterized queries** – Values in parameterized queries (p. 370) can now be cast to the DOUBLE data type.

• **Apache Iceberg** – Write operations to Iceberg tables (p. 503) now succeed when Object Lock is enabled on an Amazon S3 bucket.

August 31, 2022

Published on 2022-08-31

Amazon Athena announces availability of Athena and its features in the Asia Pacific (Jakarta) Region.

This release expands Athena's availability in Asia Pacific to include Asia Pacific (Hong Kong), Asia Pacific (Jakarta), Asia Pacific (Mumbai), Asia Pacific (Osaka), Asia Pacific (Seoul), Asia Pacific (Singapore), Asia Pacific (Sydney), and Asia Pacific (Tokyo). For a complete list of AWS services available in these and other Regions, refer to the AWS Regional Services List.

August 23, 2022

Published on 2022-08-23

Release v2022.32.1 of the Athena Query Federation SDK includes the following changes:

• Added support to the Amazon Athena Oracle data source connector for SSL based connections to Amazon RDS instances. Support is limited to the Transport Layer Security (TLS) protocol and to authentication of the server by the client. Because mutual authentication it is not supported in Amazon RDS, the update does not include support for mutual authentication.

For more information, see Amazon Athena Oracle connector (p. 122).

August 3, 2022

Published on 2022-08-03

Athena releases JDBC driver version 2.0.32. The JDBC 2.0.32 driver includes the following changes:
• The User-Agent string sent to the Athena SDK has been extended to contain the driver version, JDBC specification version, and the name of the authentication plugin.
• Fixed a NullPointerException that was thrown when no value was provided for the CheckNonProxyHost parameter.
• Fixed an issue with login_url parsing in the BrowserSaml authentication plugin.
• Fixed a proxy host issue that occurred when the UseProxyforIdp parameter was set to true.

For more information, and to download the new drivers, release notes, and documentation, see Using Athena with the JDBC driver (p. 183).

August 1, 2022

Published on 2022-08-01

Athena announces improvements to the Athena Query Federation SDK and Athena prebuilt data source connectors. The improvements include the following:

• Struct parsing – Fixed a GlueFieldLexer parsing issue in the Athena Query Federation SDK that prevented certain complicated structs from displaying all of their data. This issue affected connectors built on the Athena Query Federation SDK.
• AWS Glue tables – Added additional support for the set and decimal column types in AWS Glue tables.
• DynamoDB connector – Added the ability to ignore casing on DynamoDB attribute names. For more information, see the disable_projection_and_casing section of the Amazon Athena DynamoDB connector documentation on GitHub.

For more information, see Release v2022.30.2 of Athena Query Federation on GitHub.

July 21, 2022

Published on 2022-07-21

You can now analyze and debug your queries using performance metrics and interactive, visual query analysis tools in the Athena console. The query performance data and execution details can help you identify bottlenecks in queries, inspect the operators and statistics for each stage of a query, trace the volume of data flowing between stages, and validate the impact of query predicates. You can now:

• Access the distributed and logical execution plan for your query in a single click.
• Explore the operations at each stage before the stage is run.
• Visualize the performance of completed queries with metrics for time spent in the queuing, planning, and execution stages.
• Get information about the number of rows and amount of source data processed and output by your query.
• See granular execution details for your queries presented in context and formatted as an interactive graph.
• Use precise, stage-level execution details to understand the flow of data through your query.
• Analyze query performance data programmatically using new APIs to get query runtime statistics, also released today.

To learn how to use these capabilities on your queries, watch the video tutorial Optimize Amazon Athena Queries with New Query Analysis Tools on the AWS YouTube channel.
You can now run parameterized queries directly from the Athena console or API without preparing SQL statements in advance.

When you run queries in the Athena console that have parameters in the form of question marks, the user interface now prompts you to enter values for the parameters directly. This eliminates the need to modify literal values in the query editor every time you want to run the query.

If you use the enhanced query execution API, you can now provide the execution parameters and their values in a single call.

For more information, see Using parameterized queries (p. 370) in this user guide and the AWS Big Data Blog post Use Amazon Athena parameterized queries to provide data as a service.

Athena announces the following fixes and improvements.

- Fixed an issue with DATE column conversion handling for SageMaker endpoints (UDF) that was causing query failures.

Athena releases JDBC driver version 2.0.31. The JDBC 2.0.31 driver includes the following changes:

- log4j dependency issue – Resolved a Cannot find driver class error message caused by a log4j dependency.

For more information, and to download the new drivers, release notes, and documentation, see Using Athena with the JDBC driver (p. 183).

Athena announces the following fixes and improvements.

- Iceberg support
  - Introduced support for cross-region queries. Now you can query Iceberg tables in an AWS Region that is different from the AWS Region that you are using.
  - Introduced support for server side encryption configuration. Now you can use SSE-S3/SSE-KMS to encrypt data from Iceberg write operations in Amazon S3.

For more information about using Apache Iceberg in Athena, see Using Iceberg tables (p. 503).
• **JDBC 2.0.30 driver release**

The JDBC 2.0.30 driver for Athena has the following improvements:

• Fixes a data race issue that affected parameterized prepared statements.
• Fixes an application start up issue that occurred in Gradle build environments.

To download the JDBC 2.0.30 driver, release notes, and documentation, see Using Athena with the JDBC driver (p. 183).

---

**May 6, 2022**

Published on 2022-05-06

Released the JDBC 2.0.29 and ODBC 1.1.17 drivers for Athena.

These drivers include the following changes:

• Updated the SAML plugin browser launch process.

For more information about these changes, and to download the new drivers, release notes, and documentation, see Using Athena with the JDBC driver (p. 183) and Connecting to Amazon Athena with ODBC (p. 185).

---

**April 22, 2022**

Published on 2022-04-22

Athena announces the following fixes and improvements.

• Fixed an issue in the partition indices and filtering feature with the partition cache that occurred when the following conditions were met:
  • The partition_filtering.enabled key was set to true in the AWS Glue table properties for a table.
  • The same table was used multiple times with different partition filter values.

---

**April 21, 2022**

Published on 2022-04-21

You can now use Amazon Athena to run federated queries on new data sources, including Google BigQuery, Azure Synapse, and Snowflake. New data source connectors include:

• **Azure Data Lake Storage (ADLS) Gen2 (p. 68)**
• **Azure Synapse (p. 72)**
• **Cloudera Hive (p. 78)**
• **Cloudera Impala (p. 83)**
• **Google BigQuery (p. 77)**
• **Hortonworks (p. 108)**
• **Microsoft SQL Server (p. 148)**
• **Oracle (p. 122)**
• **SAP HANA (Express Edition) (p. 138)**
For a complete list of data sources supported by Athena, see Using Athena data source connectors (p. 67).

To make it easier to browse the available sources and connect to your data, you can now search, sort, and filter the available connectors from an updated Data Sources screen in the Athena console.

To learn about querying federated sources, see Using Amazon Athena Federated Query (p. 60) and Writing federated queries (p. 167).

**April 13, 2022**

Published on 2022-04-13

Athena releases JDBC driver version 2.0.28. The JDBC 2.0.28 driver includes the following changes:

- **JWT support** – The driver now supports JSON web tokens (JWT) for authentication. For information about using JWT with the JDBC driver, see the installation and configuration guide, downloadable from the JDBC driver page (p. 183).
- **Updated Log4j libraries** – The JDBC driver now uses the following Log4j libraries:
  - Log4j-api 2.17.1 (previously 2.17.0)
  - Log4j-core 2.17.1 (previously 2.17.0)
  - Log4j-jcl 2.17.2
- **Other improvements** – The new driver also includes the following improvements and bug fixes:
  - The Athena prepared statements feature is now available through JDBC. For information about prepared statements, see Using parameterized queries (p. 370).
  - Athena JDBC SAML federation is now functional for the China Regions.
  - Additional minor improvements.

For more information, and to download the new drivers, release notes, and documentation, see Using Athena with the JDBC driver (p. 183).

**March 30, 2022**

Published on 2022-03-30

Athena announces the following fixes and improvements.

- **Cross-region querying** – You can now use Athena to query data located in an Amazon S3 bucket across AWS Regions including Asia Pacific (Hong Kong), Middle East (Bahrain), Africa (Cape Town), and Europe (Milan).
  - For a list of AWS Regions in which Athena is available, see Amazon Athena endpoints and quotas.
  - For information about enabling an AWS Region that is disabled by default, see Enabling a Region.
  - For information about querying across Regions, see Querying across regions (p. 442).

**March 18, 2022**

Published on 2022-03-18

Athena announces the following fixes and improvements.

- Snowflake (p. 143)
- Teradata (p. 152)

For a complete list of data sources supported by Athena, see Using Athena data source connectors (p. 67).

To make it easier to browse the available sources and connect to your data, you can now search, sort, and filter the available connectors from an updated Data Sources screen in the Athena console.

To learn about querying federated sources, see Using Amazon Athena Federated Query (p. 60) and Writing federated queries (p. 167).
- **Dynamic filtering** – Dynamic filtering (p. 814) has been improved for integer columns by efficiently applying the filter to each record of a corresponding table.
- **Iceberg** – Fixed an issue that caused failures when writing iceberg Parquet files larger than 2GB.
- **Uncompressed output** – CREATE TABLE (p. 740) statements now support writing uncompressed files. To write uncompressed files, use the following syntax:
  - CREATE TABLE (text file or JSON) – In TBLPROPERTIES, specify write.compression = NONE.
  - CREATE TABLE (Parquet) – In TBLPROPERTIES, specify parquet.compression = UNCOMPRESSED.
  - CREATE TABLE (ORC) – In TBLPROPERTIES, specify orc.compress = NONE.
- **Compression** – Fixed an issue with inserts for text file tables that created files compressed in one format but used another compression format file extension when non-default compression methods were used.
- **Avro** – Fixed issues that occurred when reading decimals of the fixed type from Avro files.

**March 2, 2022**

Published on 2022-03-02

Athena announces the following features and improvements.

- You can now grant the Amazon S3 bucket owner full control access over query results when ACLs are enabled for the query result bucket. For more information, see Specifying a query result location (p. 344).
- You can now update existing named queries. For more information, see Using saved queries (p. 368).

**February 23, 2022**

Published on 2022-02-23

Athena announces the following fixes and performance improvements.

- Memory handling improvements to enhance performance and reduce memory errors.
- Athena now reads ORC timestamp columns with time zone information stored in stripe footers and writes ORC files with time zone (UTC) in footers. This only impacts the behavior of ORC timestamp reads if the ORC file to be read was created in a non-UTC time zone environment.
- Fixed incorrect symlink table size estimates that resulted in suboptimal query plans.
- Lateral exploded views can now be queried in the Athena console from Hive metastore data sources.
- Improved Amazon S3 read error messages to include more detailed Amazon S3 error code information.
- Fixed an issue that caused output files in ORC format to become incompatible with Apache Hive 3.1.
- Fixed an issue that caused table names with quotes to fail in certain DML and DDL queries.

**February 15, 2022**

Published on 2022-02-15

Amazon Athena has increased the active DML query quota in all AWS Regions. Active queries include both running and queued queries. With this change, you can now have more DML queries in an active state than before.

For information about Athena service quotas, see Service Quotas (p. 785). For the query quotas in the Region where you use Athena, see Amazon Athena endpoints and quotas in the AWS General Reference.
To monitor your quota usage, you can use CloudWatch usage metrics. Athena publishes the ActiveQueryCount metric in the AWS/Usage namespace. For more information, see Monitoring Athena usage metrics (p. 654).

After reviewing your usage, you can use the Service Quotas console to request a quota increase. If you previously requested a quota increase for your account, your requested quota still applies if it exceeds the new default active DML query quota. Otherwise, all accounts use the new default.

February 14, 2022

Published on 2022-02-14

This release adds the ErrorType subfield to the AthenaError response object in the Athena GetQueryExecution API action.

While the existing ErrorCategory field indicates the general source of a failed query (system, user, or other), the new ErrorType field provides more granular information about the error that occurred. Combine the information from both fields to gain insight into the causes of query failure.

For more information, see Athena error catalog (p. 777).

February 9, 2022

Published on 2022-02-09

The old Athena console is no longer available. Athena's new console supports all of the features of the earlier console, but with an easier to use, modern interface and includes new features that improve the experience of developing queries, analyzing data, and managing your usage. To use the new Athena console, visit https://console.aws.amazon.com/athena/.

February 8, 2022

Published on 2022-02-08

Expected bucket owner – As an added security measure, you can now optionally specify the AWS account ID that you expect to be the owner of your query results output location bucket in Athena. If the account ID of the query results bucket owner does not match the account ID that you specify, attempts to output to the bucket will fail with an Amazon S3 permissions error. You can make this setting at the client or workgroup level.

For more information, see Specifying a query result location (p. 344).

January 28, 2022

Published on 2022-01-28

Athena announces the following engine feature enhancements.

- **Apache Hudi** – Snapshot queries on Hudi Merge On Read (MoR) tables can now read timestamp columns that have the INT64 data type.
- **UNION queries** – Performance improvement and data scan reduction for certain UNION queries that scan the same table multiple times.
- **Disjunct queries** – Performance improvement for queries that have only disjunct values for each partition column on the filter.
- **Partition projection enhancements**
• Multiple disjunct values are now allowed on the filter condition for columns of the *injected* type. For more information, see *Injected type* (p. 274).
• Performance improvement for columns of string-based types like CHAR or VARCHAR that have only disjunct values on the filter.

### January 13, 2022

Published on 2022-01-13

Released the JDBC 2.0.27 and ODBC 1.1.15 drivers for Athena.

The JDBC 2.0.27 driver includes the following changes:

• The driver has been updated to retrieve external catalogs.
• The extended driver version number is now included in the *user-agent* string as part of the Athena API call.

The ODBC 1.1.15 driver includes the following changes:

• Corrects an issue with second calls to `SQLParamData()`.

For more information about these changes, and to download the new drivers, release notes, and documentation, see *Using Athena with the JDBC driver* (p. 183) and *Connecting to Amazon Athena with ODBC* (p. 185).

### Athena release notes for 2021

#### November 30, 2021

Published on 2021-11-30

Amazon Athena users can now use AWS Lake Formation to configure fine-grained access permissions and read from ACID-compliant tables. Amazon Athena makes it simple to analyze data in Amazon S3 based data lakes to help ensure that users only have access to data to which they're authorized. The ACID features help ensure that their queries are reliable in the face of complex changes to the underlying data.

Use [Lake Formation data filtering](https://docs.aws.amazon.com/athena/latest/ug/lake-formation-data-filtering.html) to secure the tables in your Amazon S3 data lake by granting permissions at the column, row, and cell levels. These permissions are enforced when Athena users query your data. This fine level of control means that you can grant users access to sensitive information without using course-grained masking that would otherwise impede their analyses. Furthermore, with Lake Formation governed tables, Athena users can query data while multiple users simultaneously add and delete the table's Amazon S3 data objects.

For more information, see *Using Athena ACID transactions* (p. 498) and *Using governed tables* (p. 498).

#### November 26, 2021

Published on 2021-11-26

Athena announces the public preview of Athena ACID transactions, which add write, delete, update, and time travel operations to Athena's SQL data manipulation language (DML). Athena ACID transactions
enable multiple concurrent users to make reliable, row-level modifications to Amazon S3 data. Built on the Apache Iceberg table format, Athena ACID transactions are compatible with other services and engines such as Amazon EMR and Apache Spark that also support the Iceberg table formats.

Athena ACID transactions and familiar SQL syntax simplify updates to your business and regulatory data. For example, to respond to a data erasure request, you can perform a SQL DELETE operation. To make manual record corrections, you can use a single UPDATE statement. To recover data that was recently deleted, you can issue time travel queries using a SELECT statement. Athena transactions are available through Athena's console, API operations, and ODBC and JDBC drivers.

For more information, see Using Athena ACID transactions (p. 498).

November 24, 2021

Published on 2021-11-24

Athena announces support for reading and writing ZStandard compressed ORC, Parquet, and textfile data. Athena uses ZStandard compression level 3 when writing ZStandard compressed data.

For information about data compression in Athena, see Athena compression support (p. 299).

November 22, 2021

Published on 2021-11-22

You can now manage AWS Step Functions workflows from the Amazon Athena console, making it easier to build scalable data processing pipelines, execute queries based on custom business logic, automate administrative and alerting tasks, and more.

Step Functions is now integrated with Athena's upgraded console, and you can use it to view an interactive workflow diagram of your state machines that invoke Athena. To get started, select Workflows from the left navigation panel. If you have existing state machines with Athena queries, select a state machine to view an interactive diagram of the workflow. If you are new to Step Functions, you can get started by launching a sample project from the Athena console and customizing it to suit your use cases.

For more information, see Build and orchestrate ETL pipelines using Amazon Athena and AWS Step Functions, or consult the Step Functions documentation.

November 18, 2021

Published on 2021-11-18

Athena announces new features and improvements.

• Support for spill-to-disk for aggregation queries that contain DISTINCT, ORDER BY, or both, as in the following example:

```sql
SELECT array_agg(orderstatus ORDER BY orderstatus)
FROM orders
GROUP BY orderpriority, custkey
```

• Addressed memory handling issues for queries that use DISTINCT. To avoid error messages like Query exhausted resources at this scale factor when you use DISTINCT queries, choose columns that have a low cardinality for DISTINCT, or reduce the data size of the query.

• In SELECT COUNT(*) queries that do not specify a specific column, improved performance and memory usage by keeping only the count without row buffering.
• Introduced the following string functions.
  • `translate(source, from, to)` – Returns the `source` string with the characters found in the `from` string replaced by the corresponding characters in the `to` string. If the `from` string contains duplicates, only the first is used. If the `source` character does not exist in the `from` string, the `source` character is copied without translation. If the index of the matching character in the `from` string is greater than the length of the `to` string, the character is omitted from the resulting string.
  • `concat_ws(string0, array(varchar))` – Returns the concatenation of elements in the array using `string0` as a separator. If `string0` is null, then the return value is null. Any null values in the array are skipped.
• Fixed a bug in which queries failed when trying to access a missing subfield in a `struct`. Queries now return a null for the missing subfield.
• Fixed an issue with inconsistent hashing for the decimal data type.
• Fixed an issue that caused exhausted resources when there were too many columns in a partition.

November 17, 2021
Published on 2021-11-17

Amazon Athena now supports partition indexing to accelerate queries on partitioned tables in the AWS Glue Data Catalog.

When querying partitioned tables, Athena retrieves and filters the available table partitions to the subset relevant to your query. As new data and partitions are added, more time is required to process the partitions and query runtime can increase. To optimize partition processing and improve query performance on highly partitioned tables, Athena now supports AWS Glue partition indexes.

For more information, see AWS Glue partition indexing and filtering (p. 29).

November 16, 2021
Published on 2021-11-16

The new and improved Amazon Athena console is now generally available in the AWS commercial and GovCloud regions where Athena is available. Athena's new console supports all of the features of the earlier console, but with an easier to use, modern interface and includes new features that improve the experience of developing queries, analyzing data, and managing your usage. You can now:
• Rearrange, navigate to, or close multiple query tabs from a redesigned query tab bar.
• Read and edit queries more easily with improved SQL and text formatting.
• Copy query results to your clipboard in addition to downloading the full result set.
• Sort your query history, saved queries, and workgroups, and choose which columns to show or hide.
• Use a simplified interface to configure data sources and workgroups in fewer clicks.
• Set preferences for displaying query results, query history, line wrapping, and more.
• Increase your productivity with new and improved keyboard shortcuts and embedded product documentation.

With today's announcement, the redesigned console is now the default. To tell us about your experience, choose Feedback in the bottom-left corner of the console.

If desired, you may use the earlier console by logging into your AWS account, choosing Amazon Athena, and deselecting New Athena experience from the navigation panel on the left.
November 12, 2021

Published on 2021-11-12

You can now use Amazon Athena to run federated queries on data sources located in an AWS account other than your own. Until today, querying this data required the data source and its connector to use the same AWS account as the user that queried the data.

As a data administrator, you can enable cross-account federated queries by sharing your data connector with a data analyst's account. As a data analyst, you can add a data connector that a data administrator has shared with you to your account. Configuration changes to the connector in the originating account apply automatically to the shared connector.

For information about enabling cross-account federated queries, see Enabling cross-account federated queries (p. 171). To learn about querying federated sources, see Using Amazon Athena Federated Query (p. 60) and Writing federated queries (p. 167).

November 2, 2021

Published on 2021-11-02

You can now use the EXPLAIN ANALYZE statement in Athena to view the distributed execution plan and cost of each operation for your SQL queries.

For more information, see Using EXPLAIN and EXPLAIN ANALYZE in Athena (p. 703).

October 29, 2021

Published on 2021-10-29

Athena releases JDBC 2.0.25 and ODBC 1.1.13 drivers and announces features and improvements.

JDBC and ODBC Drivers

Released JDBC 2.0.25 and ODBC 1.1.13 drivers for Athena. Both drivers offer support for browser SAML multi-factor authentication, which can be configured to work with any SAML 2.0 provider.

The JDBC 2.0.25 driver includes the following changes:

- Support for browser SAML authentication. The driver includes a browser SAML plugin which can be configured to work with any SAML 2.0 provider.
- Support for AWS Glue API calls. You can use the GlueEndpointOverride parameter to override the AWS Glue endpoint.
- Changed the com.simba.athena.amazonaws class path to com.amazonaws.

The ODBC 1.1.13 driver includes the following changes:

- Support for browser SAML authentication. The driver includes a browser SAML plugin which can be configured to work with any SAML 2.0 provider. For an example of how to use the browser SAML plugin with the ODBC driver, see Configuring single sign-on using ODBC, SAML 2.0, and the Okta Identity Provider (p. 204).
- You can now configure the role session duration when you use ADFS, Azure AD, or Browser Azure AD for authentication.
Features and Improvements

Athena announces the following features and improvements.

• A new optimization rule has been introduced to avoid duplicate table scans in certain cases.

October 4, 2021

Published on 2021-10-04

Athena announces the following features and improvements.

• SQL OFFSET – The SQL OFFSET clause is now supported in SELECT statements. For more information, see SELECT (p. 692).
• CloudWatch usage metrics – Athena now publishes the ActiveQueryCount metric in the AWS/Usage namespace. For more information, see Monitoring Athena usage metrics (p. 654).
• Query planning – Fixed a bug that could in rare cases cause query planning timeouts.

September 16, 2021

Published on 2021-09-16

Athena announces the following new features and improvements.

Features

• The Apache Hudi Metadata Listing Feature is now available for Hudi tables, reducing Amazon S3 overhead and query times for partitioned table queries. For information about using Apache Hudi in Athena, see Using Athena to query Apache Hudi datasets (p. 420).
• Added support for specifying textfile and JSON compression in CTAS using the write_compression table property. You can also specify the write_compression property in CTAS for the Parquet and ORC formats. For more information, see CTAS table properties (p. 745).
• The BZIP2 compression format is now supported for writing textfile and JSON files. For more information about the compression formats in Athena, see Athena compression support (p. 299).

Improvements

• Fixed a bug in which identity information failed to be sent to the UDF Lambda function.
• Fixed a predicate pushdown issue with disjunct filter conditions.
• Fixed a hashing issue for decimal types.
• Fixed an unnecessary statistics collection issue.
• Removed an inconsistent error message.
• Improved broadcast join performance by applying dynamic partition pruning in the worker node.
• For federated queries:
  • Altered configuration to reduce the occurrence of CONSTRAINT_VIOLATION errors in federated queries.

For more information about these and other changes, and to download the new drivers, release notes, and documentation, see Using Athena with the JDBC driver (p. 183) and Connecting to Amazon Athena with ODBC (p. 185).
September 15, 2021
Published on 2021-09-15

You can now use a redesigned Amazon Athena console (Preview). A new Athena JDBC driver has been released.

Athena Console Preview

You can now use a redesigned Amazon Athena console (Preview) from any AWS Region where Athena is available. The new console supports all of the features of the existing console, but from an easier to use, modern interface.

To switch to the new console, log into your AWS account and choose Amazon Athena. From the AWS console navigation bar, choose Switch to the new console. To return to the default console, deselect New Athena experience from the navigation panel on the left.

Get started with the new console today. Choose Feedback in the bottom-left corner to tell us about your experience.

Athena JDBC Driver 2.0.24

Athena announces availability of JDBC driver version 2.0.24 for Athena. This release updates proxy support for all credentials providers. The driver now supports proxy authentication for all hosts that are not supported by the NonProxyHosts connection property.

As a convenience, this release includes downloads of the JDBC driver both with and without the AWS SDK. This JDBC driver version allows you to have both the AWS-SDK and the Athena JDBC driver embedded in project.

For more information and to download the new driver, release notes, and documentation, see Using Athena with the JDBC driver (p. 183).

August 31, 2021
Published on 2021-08-31

Athena announces the following feature enhancements and bug fixes.

- Athena federation enhancements – Athena has added support to map types and better support for complex types as part of the Athena Query Federation SDK. This version also includes some memory enhancements and performance optimizations.
- New error categories – Introduced the USER and SYSTEM error categories in error messages. These categories help you distinguish between errors that you can fix yourself (USER) and errors that can require assistance from Athena support (SYSTEM).
- Federated query error messaging – Updated USER_ERROR categorizations for federated query related errors.
- JOIN – Fixed spill-to-disk related bugs and memory issues to enhance performance and reduce memory errors in JOIN operations.

August 12, 2021
Published on 2021-08-12

Released the ODBC 1.1.12 driver for Athena. This version corrects issues related to SQLPrepare(), SQLGetInfo(), and EndpointOverride.
To download the new driver, release notes, and documentation, see Connecting to Amazon Athena with ODBC (p. 185).

August 6, 2021

Published on 2021-08-06

Amazon Athena announces availability of Athena and its features in the Asia Pacific (Osaka) Region.

This release expands Athena’s availability in Asia Pacific to include Asia Pacific (Hong Kong), Asia Pacific (Mumbai), Asia Pacific (Osaka), Asia Pacific (Seoul), Asia Pacific (Singapore), Asia Pacific (Sydney), and Asia Pacific (Tokyo). For a complete list of AWS services available in these and other Regions, refer to the AWS Regional Services List.

August 5, 2021

Published on 2021-08-05

You can use the UNLOAD statement to write the output of a SELECT query to the PARQUET, ORC, AVRO, and JSON formats.

For more information, see UNLOAD (p. 717).

July 30, 2021

Published on 2021-07-30

Athena announces the following feature enhancements and bug fixes.

- **Dynamic filtering and partition pruning** – Improvements increase performance and reduce the amount of data scanned in certain queries, as in the following example.

  This example assumes that Table_B is an unpartitioned table that has file sizes that add up to less than 20Mb. For queries like this, less data is read from Table_A and the query completes more quickly.

  ```sql
  SELECT *
  FROM Table_A
  JOIN Table_B ON Table_A.date = Table_B.date
  WHERE Table_B.column_A = "value"
  ```

- **ORDER BY with LIMIT, DISTINCT with LIMIT** – Performance improvements to queries that use ORDER BY or DISTINCT followed by a LIMIT clause.

- **S3 Glacier Deep Archive files** – When Athena queries a table that contains a mix of S3 Glacier Deep Archive files and non-S3 Glacier files, Athena now skips the S3 Glacier Deep Archive files for you. Previously, you were required to manually move these files from the query location or the query would fail. If you want to use Athena to query objects in S3 Glacier Deep Archive storage, you must restore them. For more information, see Restoring an archived object in the Amazon S3 User Guide.

- Fixed a bug in which empty files created by the CTAS bucketed_by table property (p. 745) were not encrypted correctly.

July 21, 2021

Published on 2021-07-21
With the July 2021 release of Microsoft Power BI Desktop, you can create reports and dashboards using a native data source connector for Amazon Athena. The connector for Amazon Athena is available as a standard connector in Power BI, supports DirectQuery, and enables analysis on large datasets and content refresh through Power BI Gateway.

Because the connector uses your existing ODBC data source name (DSN) to connect to and run queries on Athena, it requires the Athena ODBC driver. To download the latest ODBC driver, see Connecting to Amazon Athena with ODBC (p. 185).

For more information, see Using the Amazon Athena Power BI connector (p. 245).

July 16, 2021

Published on 2021-07-16

Amazon Athena has updated its integration with Apache Hudi. Hudi is an open-source data management framework used to simplify incremental data processing in Amazon S3 data lakes. The updated integration enables you to use Athena to query Hudi 0.8.0 tables managed through Amazon EMR, Apache Spark, Apache Hive or other compatible services. In addition, Athena now supports two additional features: snapshot queries on Merge-on-Read (MoR) tables and read support on bootstrapped tables.

Apache Hudi provides record-level data processing that can help you simplify development of Change Data Capture (CDC) pipelines, comply with GDPR-driven updates and deletes, and better manage streaming data from sensors or devices that require data insertion and event updates. The 0.8.0 release makes it easier to migrate large Parquet tables to Hudi without copying data so you can query and analyze them through Athena. You can use Athena's new support for snapshot queries to have near real-time views of your streaming table updates.

To learn more about using Hudi with Athena, see Using Athena to query Apache Hudi datasets (p. 420).

July 8, 2021

Published on 2021-07-08

Released the ODBC 1.1.11 driver for Athena. The ODBC driver can now authenticate the connection using a JSON Web Token (JWT). On Linux, the default value for the Workgroup property has been set to Primary.

For more information and to download the new driver, release notes, and documentation, see Connecting to Amazon Athena with ODBC (p. 185).

July 1, 2021

Published on 2021-07-01

On July 1, 2021, special handling of preview workgroups ended. While AmazonAthenaPreviewFunctionality workgroups retain their name, they no longer have special status. You can continue to use AmazonAthenaPreviewFunctionality workgroups to view, modify, organize, and run queries. However, queries that use features that were formerly in preview are now subject to standard Athena billing terms and conditions. For billing information, see Amazon Athena pricing.

June 23, 2021

Published on 2021-06-23
Released JDBC 2.0.23 and ODBC 1.1.10 drivers for Athena. Both drivers offer improved read performance and support EXPLAIN (p. 703) statements and parameterized queries (p. 370).

EXPLAIN statements show the logical or distributed execution plan of a SQL query. Parameterized queries enable the same query to be used multiple times with different values supplied at run time.

The JDBC release also adds support for Active Directory Federation Services 2019 and a custom endpoint override option for AWS STS. The ODBC release fixes an issue with IAM profile credentials.

For more information and to download the new drivers, release notes, and documentation, see Using Athena with the JDBC driver (p. 183) and Connecting to Amazon Athena with ODBC (p. 185).

May 12, 2021

Published on 2021-05-12

You can now use Amazon Athena to register an AWS Glue catalog from an account other than your own. After you configure the required IAM permissions for AWS Glue, you can use Athena to run cross-account queries.

For more information, see Registering an AWS Glue Data Catalog from another account (p. 24) and Cross-account access to AWS Glue data catalogs (p. 548).

May 10, 2021

Published on 2021-05-10

Released ODBC driver version 1.1.9.1001 for Athena. This version fixes an issue with the BrowserAzureAD authentication type when using Azure Active Directory (AD).

To download the new drivers, release notes, and documentation, see Connecting to Amazon Athena with ODBC (p. 185).

May 5, 2021

Published on 2021-05-05

You can now use the Amazon Athena Vertica connector in federated queries to query Vertica data sources from Athena. For example, you can run analytical queries over a data warehouse on Vertica and a data lake in Amazon S3.

To deploy the Athena Vertica connector, visit the AthenaVerticaConnector page in the AWS Serverless Application Repository.

The Amazon Athena Vertica connector exposes several configuration options through Lambda environment variables. For information about configuration options, parameters, connection strings, deployment, and limitations, see Amazon Athena Vertica connector (p. 163).

For in-depth information about using the Vertica connector, see Querying a Vertica data source in Amazon Athena using the Athena Federated Query SDK in the AWS Big Data Blog.

April 30, 2021

Published on 2021-04-30

Released drivers JDBC 2.0.21 and ODBC 1.1.9 for Athena. Both releases support SAML authentication with Azure Active Directory (AD) and SAML authentication with PingFederate. The JDBC release also
supports parameterized queries. For information about parameterized queries in Athena, see Using parameterized queries (p. 370).

To download the new drivers, release notes, and documentation, see Using Athena with the JDBC driver (p. 183) and Connecting to Amazon Athena with ODBC (p. 185).

April 29, 2021
Published on 2021-04-29
Amazon Athena announces availability of Athena engine version 2 in the China (Beijing) and China (Ningxia) Regions.
For information about Athena engine version 2, see Athena engine version 2 (p. 680).

April 26, 2021
Published on 2021-04-26
Window value functions in Athena engine version 2 now support IGNORE NULLS and RESPECT NULLS.
For more information, see Value Functions in the Presto documentation.

April 21, 2021
Published on 2021-04-21
Amazon Athena announces availability of Athena engine version 2 in the Europe (Milan) and Africa (Cape Town) Regions.
For information about Athena engine version 2, see Athena engine version 2 (p. 680).

April 5, 2021
Published on 2021-04-05
EXPLAIN Statement
You can now use the EXPLAIN statement in Athena to view the execution plan for your SQL queries.
For more information, see Using EXPLAIN and EXPLAIN ANALYZE in Athena (p. 703) and Understanding Athena EXPLAIN statement results (p. 712).

SageMaker Machine Learning Models in SQL Queries
Machine learning model inference with Amazon SageMaker is now generally available for Amazon Athena. Use machine learning models in SQL queries to simplify complex tasks such as anomaly detection, customer cohort analysis, and time-series predictions by invoking a function in a SQL query.
For more information, see Using Machine Learning (ML) with Amazon Athena (p. 432).

User Defined Functions (UDF)
User defined functions (UDFs) are now generally available for Athena. Use UDFs to leverage custom functions that process records or groups of records in a single SQL query.
For more information, see Querying with user defined functions (p. 434).
March 30, 2021
Published on 2021-03-30
Amazon Athena announces availability of Athena engine version 2 in the Asia Pacific (Hong Kong) and Middle East (Bahrain) Regions.
For information about Athena engine version 2, see Athena engine version 2 (p. 680).

March 25, 2021
Published on 2021-03-25
Amazon Athena announces availability of Athena engine version 2 in the Europe (Stockholm) Region.
For information about Athena engine version 2, see Athena engine version 2 (p. 680).

March 5, 2021
Published on 2021-03-05
Amazon Athena announces availability of Athena engine version 2 in the Canada (Central), Europe (Frankfurt), and South America (São Paulo) Regions.
For information about Athena engine version 2, see Athena engine version 2 (p. 680).

February 25, 2021
Published on 2021-02-25
Amazon Athena announces general availability of Athena engine version 2 in the Asia Pacific (Seoul), Asia Pacific (Singapore), Asia Pacific (Sydney), Europe (London), and Europe (Paris) Regions.
For information about Athena engine version 2, see Athena engine version 2 (p. 680).

Athena release notes for 2020

December 16, 2020
Published on 2020-12-16
Amazon Athena announces availability of Athena engine version 2, Athena Federated Query, and AWS PrivateLink in additional Regions.

Athena engine version 2 and Athena Federated Query
Amazon Athena announces general availability of Athena engine version 2 and Athena Federated Query in the Asia Pacific (Mumbai), Asia Pacific (Tokyo), Europe (Ireland), and US West (N. California) Regions. Athena engine version 2 and federated queries are already available in the US East (N. Virginia), US East (Ohio), and US West (Oregon) Regions.
For more information, see Athena engine version 2 (p. 680) and Using Amazon Athena Federated Query (p. 60).
AWS PrivateLink

AWS PrivateLink for Athena is now supported in the Europe (Stockholm) Region. For information about AWS PrivateLink for Athena, see Connect to Amazon Athena using an interface VPC endpoint (p. 579).

November 24, 2020

Published on 2020-11-24

Released drivers JDBC 2.0.16 and ODBC 1.1.6 for Athena. These releases support, at the account level, Okta Verify multifactor authentication (MFA). You can also use Okta MFA to configure SMS authentication and Google Authenticator authentication as factors.

To download the new drivers, release notes, and documentation, see Using Athena with the JDBC driver (p. 183) and Connecting to Amazon Athena with ODBC (p. 185).

November 11, 2020

Published on 2020-11-11

Amazon Athena announces general availability in the US East (N. Virginia), US East (Ohio), and US West (Oregon) Regions for Athena engine version 2 and federated queries.

Athena engine version 2

Amazon Athena announces general availability of a new query engine version, Athena engine version 2, in the US East (N. Virginia), US East (Ohio), and US West (Oregon) Regions.

Athena engine version 2 includes performance enhancements and new feature capabilities such as schema evolution support for Parquet format data, additional geospatial functions, support for reading nested schema to reduce cost, and performance enhancements in JOIN and AGGREGATE operations.

- For information about improvements, breaking changes, and bug fixes, see Athena engine version 2 (p. 680).
- For information about how to upgrade, see Changing Athena engine versions (p. 669).
- For information about testing queries, see Testing queries in advance of an engine version upgrade (p. 671).

Federated SQL Queries

You can now use Athena's federated query in the US East (N. Virginia), US East (Ohio), and US West (Oregon) Regions without using the AmazonAthenaPreviewFunctionality workgroup.

Use Federated SQL queries to run SQL queries across relational, non-relational, object, and custom data sources. With federated querying, you can submit a single SQL query that scans data from multiple sources running on premises or hosted in the cloud.

Running analytics on data spread across applications can be complex and time consuming for the following reasons:

- Data required for analytics is often spread across relational, key-value, document, in-memory, search, graph, object, time-series and ledger data stores.
- To analyze data across these sources, analysts build complex pipelines to extract, transform, and load into a data warehouse so that the data can be queried.
- Accessing data from various sources requires learning new programming languages and data access constructs.
Federated SQL queries in Athena eliminate this complexity by allowing users to query the data in-place from wherever it resides. Analysts can use familiar SQL constructs to JOIN data across multiple data sources for quick analysis, and store results in Amazon S3 for subsequent use.

Data Source Connectors

To process federated queries, Athena uses Athena Data Source Connectors that run on AWS Lambda. The following open sourced, pre-built connectors were written and tested by Athena. Use them to run SQL queries in Athena on their corresponding data sources.

- CloudWatch (p. 87)
- CloudWatch Metrics (p. 89)
- DocumentDB (p. 97)
- DynamoDB (p. 101)
- OpenSearch (p. 118)
- HBase (p. 104)
- Neptune (p. 117)
- Redis (p. 131)
- Timestream (p. 157)
- TPC Benchmark DS (TPC-DS) (p. 160)

Custom Data Source Connectors

Using Athena Query Federation SDK, developers can build connectors to any data source to enable Athena to run SQL queries against that data source. Athena Query Federation Connector extends the benefits of federated querying beyond AWS provided connectors. Because connectors run on AWS Lambda, you do not have to manage infrastructure or plan for scaling to peak demands.

Next Steps

- To learn more about the federated query feature, see Using Amazon Athena Federated Query (p. 60).
- To get started with using an existing connector, see Deploying a Connector and Connecting to a Data Source.
- To learn how to build your own data source connector using the Athena Query Federation SDK, see Example Athena Connector on GitHub.

October 22, 2020

Published on 2020-10-22

You can now call Athena with AWS Step Functions. AWS Step Functions can control certain AWS services directly using the Amazon States Language. You can use Step Functions with Athena to start and stop query execution, get query results, run ad-hoc or scheduled data queries, and retrieve results from data lakes in Amazon S3.

For more information, see Call Athena with Step Functions in the AWS Step Functions Developer Guide.

July 29, 2020

Published on 2020-07-29

Released JDBC driver version 2.0.13. This release supports using multiple data catalogs registered with Athena (p. 62), Okta service for authentication, and connections to VPC endpoints.
Amazon Athena adds support for querying compacted Hudi datasets and adds the AWS CloudFormation `AWS::Athena::DataCatalog` resource for creating, updating, or deleting data catalogs that you register in Athena.

Querying Apache Hudi Datasets

Apache Hudi is an open-source data management framework that simplifies incremental data processing. Amazon Athena now supports querying the read-optimized view of an Apache Hudi dataset in your Amazon S3-based data lake.

For more information, see Using Athena to query Apache Hudi datasets (p. 420).

AWS CloudFormation Data Catalog Resource

To use Amazon Athena’s federated query feature to query any data source, you must first register your data catalog in Athena. You can now use the AWS CloudFormation `AWS::Athena::DataCatalog` resource to create, update, or delete data catalogs that you register in Athena.

For more information, see `AWS::Athena::DataCatalog` in the AWS CloudFormation User Guide.

Using Apache Hive Metastore as a Metacatalog with Amazon Athena

You can now connect Athena to one or more Apache Hive metastores in addition to the AWS Glue Data Catalog with Athena.

To connect to a self-hosted Hive metastore, you need an Athena Hive metastore connector. Athena provides a reference implementation (p. 59) connector that you can use. The connector runs as an AWS Lambda function in your account.

For more information, see Using Athena Data Connector for External Hive Metastore (p. 35).

Amazon Athena adds support for partition projection. Use partition projection to speed up query processing of highly partitioned tables and automate partition management. For more information, see Partition projection with Amazon Athena (p. 265).
In addition to the US East (N. Virginia) Region, the Amazon Athena federated query (p. 60), user defined functions (UDFs) (p. 434), machine learning inference (p. 432), and external Hive metastore (p. 35) features are now available in preview in the Asia Pacific (Mumbai), Europe (Ireland), and US West (Oregon) Regions.

March 11, 2020

Published on 2020-03-11

Amazon Athena now publishes Amazon CloudWatch Events for query state transitions. When a query transitions between states -- for example, from Running to a terminal state such as Succeeded or Cancelled -- Athena publishes a query state change event to CloudWatch Events. The event contains information about the query state transition. For more information, see Monitoring Athena queries with CloudWatch events (p. 651).

March 6, 2020

Published on 2020-03-06

You can now create and update Amazon Athena workgroups by using the AWS CloudFormation AWS::Athena::WorkGroup resource. For more information, see AWS::Athena::WorkGroup in the AWS CloudFormation User Guide.

Athena release notes for 2019

November 26, 2019

Published on 2019-12-17

Amazon Athena adds support for running SQL queries across relational, non-relational, object, and custom data sources, invoking machine learning models in SQL queries, User Defined Functions (UDFs) (Preview), using Apache Hive Metastore as a metadata catalog with Amazon Athena (Preview), and four additional query-related metrics.

Federated SQL Queries

Use Federated SQL queries to run SQL queries across relational, non-relational, object, and custom data sources.

You can now use Athena's federated query to scan data stored in relational, non-relational, object, and custom data sources. With federated querying, you can submit a single SQL query that scans data from multiple sources running on premises or hosted in the cloud.

Running analytics on data spread across applications can be complex and time consuming for the following reasons:

- Data required for analytics is often spread across relational, key-value, document, in-memory, search, graph, object, time-series and ledger data stores.
- To analyze data across these sources, analysts build complex pipelines to extract, transform, and load into a data warehouse so that the data can be queried.
- Accessing data from various sources requires learning new programming languages and data access constructs.
Federated SQL queries in Athena eliminate this complexity by allowing users to query the data in-place from wherever it resides. Analysts can use familiar SQL constructs to JOIN data across multiple data sources for quick analysis, and store results in Amazon S3 for subsequent use.

**Data Source Connectors**

Athena processes federated queries using Athena Data Source Connectors that run on AWS Lambda. Use these open sourced data source connectors to run federated SQL queries in Athena across Amazon DynamoDB, Apache HBase, Amazon Document DB, Amazon CloudWatch, Amazon CloudWatch Metrics, and JDBC-compliant relational databases such MySQL, and PostgreSQL under the Apache 2.0 license.

**Custom Data Source Connectors**

Using Athena Query Federation SDK, developers can build connectors to any data source to enable Athena to run SQL queries against that data source. Athena Query Federation Connector extends the benefits of federated querying beyond AWS provided connectors. Because connectors run on AWS Lambda, you do not have to manage infrastructure or plan for scaling to peak demands.

**Preview Availability**

Athena federated query is available in preview in the US East (N. Virginia) Region.

**Next Steps**

- To begin your preview, follow the instructions in the Athena Preview Features FAQ.
- To learn more about the federated query feature, see Using Amazon Athena Federated Query (Preview).
- To get started with using an existing connector, see Deploying a Connector and Connecting to a Data Source.
- To learn how to build your own data source connector using the Athena Query Federation SDK, see Example Athena Connector on GitHub.

**Invoking Machine Learning Models in SQL Queries**

You can now invoke machine learning models for inference directly from your Athena queries. The ability to use machine learning models in SQL queries makes complex tasks such anomaly detection, customer cohort analysis, and sales predictions as simple as invoking a function in a SQL query.

**ML Models**

You can use more than a dozen built-in machine learning algorithms provided by Amazon SageMaker, train your own models, or find and subscribe to model packages from AWS Marketplace and deploy on Amazon SageMaker Hosting Services. There is no additional setup required. You can invoke these ML models in your SQL queries from the Athena console, Athena APIs, and through Athena's preview JDBC driver.

**Preview Availability**

Athena's ML functionality is available today in preview in the US East (N. Virginia) Region.

**Next Steps**

- To begin your preview, follow the instructions in the Athena Preview Features FAQ.
- To learn more about the machine learning feature, see Using Machine Learning (ML) with Amazon Athena (Preview).
User Defined Functions (UDFs) (Preview)

You can now write custom scalar functions and invoke them in your Athena queries. You can write your UDFs in Java using the Athena Query Federation SDK. When a UDF is used in a SQL query submitted to Athena, it is invoked and run on AWS Lambda. UDFs can be used in both SELECT and FILTER clauses of a SQL query. You can invoke multiple UDFs in the same query.

Preview Availability

Athena UDF functionality is available in Preview mode in the US East (N. Virginia) Region.

Next Steps

- To begin your preview, follow the instructions in the Athena Preview Features FAQ.
- To learn more, see Querying with User Defined Functions (Preview).
- For example UDF implementations, see Amazon Athena UDF Connector on GitHub.
- To learn how to write your own functions using the Athena Query Federation SDK, see Creating and Deploying a UDF Using Lambda.

Using Apache Hive Metastore as a Metacatalog with Amazon Athena (Preview)

You can now connect Athena to one or more Apache Hive Metastores in addition to the AWS Glue Data Catalog with Athena.

Metastore Connector

To connect to a self-hosted Hive Metastore, you need an Athena Hive Metastore connector. Athena provides a reference implementation connector that you can use. The connector runs as an AWS Lambda function in your account. For more information, see Using Athena Data Connector for External Hive Metastore (Preview).

Preview Availability

The Hive Metastore feature is available in Preview mode in the US East (N. Virginia) Region.

Next Steps

- To begin your preview, follow the instructions in the Athena Preview Features FAQ.
- To learn more about this feature, please visit our Using Athena Data Connector for External Hive Metastore (Preview).

New Query-Related Metrics

Athena now publishes additional query metrics that can help you understand Amazon Athena performance. Athena publishes query-related metrics to Amazon CloudWatch. In this release, Athena publishes the following additional query metrics:

- **Query Planning Time** – The time taken to plan the query. This includes the time spent retrieving table partitions from the data source.
- **Query Queuing Time** – The time that the query was in a queue waiting for resources.
- **Service Processing Time** – The time taken to write results after the query engine finishes processing.
- **Total Execution Time** – The time Athena took to run the query.
To consume these new query metrics, you can create custom dashboards, set alarms and triggers on metrics in CloudWatch, or use pre-populated dashboards directly from the Athena console.

**Next Steps**

For more information, see [Monitoring Athena Queries with CloudWatch Metrics](#).

---

**November 12, 2019**

Published on 2019-12-17

Amazon Athena is now available in the Middle East (Bahrain) Region.

**November 8, 2019**

Published on 2019-12-17

Amazon Athena is now available in the US West (N. California) Region and the Europe (Paris) Region.

**October 8, 2019**

Published on 2019-12-17

Amazon Athena now allows you to connect directly to Athena through an interface VPC endpoint in your Virtual Private Cloud (VPC). Using this feature, you can submit your queries to Athena securely without requiring an Internet Gateway in your VPC.

To create an interface VPC endpoint to connect to Athena, you can use the AWS Management Console or AWS Command Line Interface (AWS CLI). For information about creating an interface endpoint, see [Creating an Interface Endpoint](#).

When you use an interface VPC endpoint, communication between your VPC and Athena APIs is secure and stays within the AWS network. There are no additional Athena costs to use this feature. Interface VPC endpoint charges apply.

To learn more about this feature, see [Connect to Amazon Athena Using an Interface VPC Endpoint](#).

**September 19, 2019**

Published on 2019-12-17

Amazon Athena adds support for inserting new data to an existing table using the `INSERT INTO` statement. You can insert new rows into a destination table based on a `SELECT` query statement that runs on a source table, or based on a set of values that are provided as part of the query statement. Supported data formats include Avro, JSON, ORC, Parquet, and text files.

`INSERT INTO` statements can also help you simplify your ETL process. For example, you can use `INSERT INTO` in a single query to select data from a source table that is in JSON format and write to a destination table in Parquet format.

`INSERT INTO` statements are charged based on the number of bytes that are scanned in the `SELECT` phase, similar to how Athena charges for `SELECT` queries. For more information, see [Amazon Athena pricing](#).

For more information about using `INSERT INTO`, including supported formats, SerDes and examples, see [INSERT INTO](#) in the Athena User Guide.
September 12, 2019
Published on 2019-12-17
Amazon Athena is now available in the Asia Pacific (Hong Kong) Region.

August 16, 2019
Published on 2019-12-17
Amazon Athena adds support for querying data in Amazon S3 Requester Pays buckets.

When an Amazon S3 bucket is configured as Requester Pays, the requester, not the bucket owner, pays for the Amazon S3 request and data transfer costs. In Athena, workgroup administrators can now configure workgroup settings to allow workgroup members to query S3 Requester Pays buckets.

For information about how to configure the Requester Pays setting for your workgroup, refer to Create a Workgroup in the Amazon Athena User Guide. For more information about Requester Pays buckets, see Requester Pays Buckets in the Amazon Simple Storage Service Developer Guide.

August 9, 2019
Published on 2019-12-17
Amazon Athena now supports enforcing AWS Lake Formation policies for fine-grained access control to new or existing databases, tables, and columns defined in the AWS Glue Data Catalog for data stored in Amazon S3.

You can use this feature in the following AWS Regions: US East (Ohio), US East (N. Virginia), US West (Oregon), Asia Pacific (Tokyo), and Europe (Ireland). There are no additional charges to use this feature.

For more information about using this feature, see Using Athena to query data registered with AWS Lake Formation (p. 580). For more information about AWS Lake Formation, see AWS Lake Formation.

June 26, 2019
Amazon Athena is now available in the Europe (Stockholm) Region. For a list of supported Regions, see AWS Regions and Endpoints.

May 24, 2019
Published on 2019-05-24
Amazon Athena is now available in the AWS GovCloud (US-East) and AWS GovCloud (US-West) Regions. For a list of supported Regions, see AWS Regions and Endpoints.

March 05, 2019
Published on 2019-03-05
Amazon Athena is now available in the Canada (Central) Region. For a list of supported Regions, see AWS Regions and Endpoints. Released the new version of the ODBC driver with support for Athena workgroups. For more information, see the ODBC Driver Release Notes.

To download the ODBC driver version 1.0.5 and its documentation, see Connecting to Amazon Athena with ODBC (p. 185). For information about this version, see the ODBC Driver Release Notes.
To use workgroups with the ODBC driver, set the new connection property, `Workgroup`, in the connection string as shown in the following example:

```
Driver=Simba Athena ODBC;
AwsRegion=[Region];
S3OutputLocation=[S3Path];
AuthenticationType=IAM;
Credentials;UID=[YourAccessKey];PWD=[YourSecretKey];Workgroup=[WorkgroupName]
```

For more information, search for "workgroup" in the ODBC Driver Installation and Configuration Guide version 1.0.5. There are no changes to the ODBC driver connection string when you use tags on workgroups. To use tags, upgrade to the latest version of the ODBC driver, which is this current version.

This driver version lets you use Athena API workgroup actions (p. 645) to create and manage workgroups, and Athena API tag actions (p. 660) to add, list, or remove tags on workgroups. Before you begin, make sure that you have resource-level permissions in IAM for actions on workgroups and tags.

For more information, see:

- Using workgroups for running queries (p. 629) and Workgroup example policies (p. 633).
- Tagging Athena resources (p. 658) and Tag-based IAM access control policies (p. 664).

If you use the JDBC driver or the AWS SDK, upgrade to the latest version of the driver and SDK, both of which already include support for workgroups and tags in Athena. For more information, see Using Athena with the JDBC driver (p. 183).

**February 22, 2019**

Added tag support for workgroups in Amazon Athena. A tag consists of a key and a value, both of which you define. When you tag a workgroup, you assign custom metadata to it. You can add tags to workgroups to help categorize them, using AWS tagging best practices. You can use tags to restrict access to workgroups, and to track costs. For example, create a workgroup for each cost center. Then, by adding tags to these workgroups, you can track your Athena spending for each cost center. For more information, see Using Tags for Billing in the AWS Billing and Cost Management User Guide.

You can work with tags by using the Athena console or the API operations. For more information, see Tagging Athena resources (p. 658).

In the Athena console, you can add one or more tags to each of your workgroups, and search by tags. Workgroups are an IAM-controlled resource in Athena. In IAM, you can restrict who can add, remove, or list tags on workgroups that you create. You can also use the CreateWorkGroup API operation that has the optional tag parameter for adding one or more tags to the workgroup. To add, remove, or list tags, use TagResource, UntagResource, and ListTagsForResource. For more information, see Using tag operations (p. 660).

To allow users to add tags when creating workgroups, ensure that you give each user IAM permissions to both the TagResource and CreateWorkGroup API actions. For more information and examples, see Tag-based IAM access control policies (p. 664).

There are no changes to the JDBC driver when you use tags on workgroups. If you create new workgroups and use the JDBC driver or the AWS SDK, upgrade to the latest version of the driver and SDK. For information, see Using Athena with the JDBC driver (p. 183).

**February 18, 2019**

Published on 2019-02-18
Added ability to control query costs by running queries in workgroups. For information, see Using workgroups to control query access and costs (p. 629). Improved the JSON OpenX SerDe used in Athena, fixed an issue where Athena did not ignore objects transitioned to the GLACIER storage class, and added examples for querying Network Load Balancer logs.

Made the following changes:

- Added support for workgroups. Use workgroups to separate users, teams, applications, or workloads, and to set limits on amount of data each query or the entire workgroup can process. Because workgroups act as IAM resources, you can use resource-level permissions to control access to a specific workgroup. You can also view query-related metrics in Amazon CloudWatch, control query costs by configuring limits on the amount of data scanned, create thresholds, and trigger actions, such as Amazon SNS alarms, when these thresholds are breached. For more information, see Using workgroups for running queries (p. 629) and Controlling costs and monitoring queries with CloudWatch metrics and events (p. 647).

Workgroups are an IAM resource. For a full list of workgroup-related actions, resources, and conditions in IAM, see Actions, Resources, and Condition Keys for Amazon Athena in the Service Authorization Reference. Before you create new workgroups, make sure that you use workgroup IAM policies (p. 632), and the AWS managed policy: AmazonAthenaFullAccess (p. 531).

You can start using workgroups in the console, with the workgroup API operations (p. 645), or with the JDBC driver. For a high-level procedure, see Setting up workgroups (p. 631). To download the JDBC driver with workgroup support, see Using Athena with the JDBC driver (p. 183).

If you use workgroups with the JDBC driver, you must set the workgroup name in the connection string using the Workgroup configuration parameter as in the following example:

```
jdbc:awsathena://<AWSREGION>;UID=<ACCESSKEY>;PWD=<SECRETKEY>;S3OutputLocation=s3://<athena-output>-<AWSREGION>;/;
Workgroup=<WORKGROUPNAME>;
```

There are no changes in the way you run SQL statements or make JDBC API calls to the driver. The driver passes the workgroup name to Athena.

For information about differences introduced with workgroups, see Athena workgroup APIs (p. 645) and Troubleshooting workgroups (p. 645).

- Improved the JSON OpenX SerDe used in Athena. The improvements include, but are not limited to, the following:
  - Support for the ConvertDotsInJsonKeysToUnderscores property. When set to TRUE, it allows the SerDe to replace the dots in key names with underscores. For example, if the JSON dataset contains a key with the name "a.b", you can use this property to define the column name to be "a_b" in Athena. The default is FALSE. By default, Athena does not allow dots in column names.
  - Support for the case.insensitive property. By default, Athena requires that all keys in your JSON dataset use lowercase. Using WITH SERDE PROPERTIES ("case.insensitive"=FALSE;) allows you to use case-sensitive key names in your data. The default is TRUE. When set to TRUE, the SerDe converts all uppercase columns to lowercase.

For more information, see OpenX JSON SerDe (p. 321).

- Fixed an issue where Athena returned "access denied" error messages, when it processed Amazon S3 objects that were archived to Glacier by Amazon S3 lifecycle policies. As a result of fixing this issue, Athena ignores objects transitioned to the GLACIER storage class. Athena does not support querying data from the GLACIER storage class.

For more information, see the section called “Requirements for tables in Athena and data in Amazon S3” (p. 252) and Transitioning to the GLACIER Storage Class (Object Archival) in the Amazon Simple Storage Service User Guide.
• Added examples for querying Network Load Balancer access logs that receive information about the Transport Layer Security (TLS) requests. For more information, see the section called “Network Load Balancer” (p. 470).

Athena release notes for 2018

November 20, 2018

Published on 2018-11-20

Released the new versions of the JDBC and ODBC driver with support for federated access to Athena API with the AD FS and SAML 2.0 (Security Assertion Markup Language 2.0). For details, see the JDBC Driver Release Notes and ODBC Driver Release Notes.

With this release, federated access to Athena is supported for the Active Directory Federation Service (AD FS 3.0). Access is established through the versions of JDBC or ODBC drivers that support SAML 2.0. For information about configuring federated access to the Athena API, see the section called “Enabling federated access to the Athena API” (p. 571).

To download the JDBC driver version 2.0.6 and its documentation, see Using Athena with the JDBC driver (p. 183). For information about this version, see JDBC Driver Release Notes.

To download the ODBC driver version 1.0.4 and its documentation, see Connecting to Amazon Athena with ODBC (p. 185). For information about this version, ODBC Driver Release Notes.

For more information about SAML 2.0 support in AWS, see About SAML 2.0 Federation in the IAM User Guide.

October 15, 2018

Published on 2018-10-15

If you have upgraded to the AWS Glue Data Catalog, there are two new features that provide support for:

• Encryption of the Data Catalog metadata. If you choose to encrypt metadata in the Data Catalog, you must add specific policies to Athena. For more information, see Access to Encrypted Metadata in the AWS Glue Data Catalog (p. 553).

• Fine-grained permissions to access resources in the AWS Glue Data Catalog. You can now define identity-based (IAM) policies that restrict or allow access to specific databases and tables from the Data Catalog used in Athena. For more information, see Fine-grained access to databases and tables in the AWS Glue Data Catalog (p. 540).

Note
Data resides in the Amazon S3 buckets, and access to it is governed by the Access to Amazon S3 (p. 536). To access data in databases and tables, continue to use access control policies to Amazon S3 buckets that store the data.

October 10, 2018

Published on 2018-10-10

Athena supports CREATE TABLE AS SELECT, which creates a table from the result of a SELECT query statement. For details, see Creating a Table from Query Results (CTAS).
Before you create CTAS queries, it is important to learn about their behavior in the Athena documentation. It contains information about the location for saving query results in Amazon S3, the list of supported formats for storing CTAS query results, the number of partitions you can create, and supported compression formats. For more information, see Considerations and limitations for CTAS queries (p. 281).

Use CTAS queries to:

- Create a table from query results (p. 281) in one step.
- Create CTAS queries in the Athena console (p. 283), using Examples (p. 286). For information about syntax, see CREATE TABLE AS (p. 744).
- Transform query results into other storage formats, such as PARQUET, ORC, AVRO, JSON, and TEXTFILE. For more information, see Considerations and limitations for CTAS queries (p. 281) and Columnar storage formats (p. 259).

September 6, 2018

Published on 2018-09-06

Released the new version of the ODBC driver (version 1.0.3). The new version of the ODBC driver streams results by default, instead of paging through them, allowing business intelligence tools to retrieve large data sets faster. This version also includes improvements, bug fixes, and an updated documentation for “Using SSL with a Proxy Server”. For details, see the Release Notes for the driver.

For downloading the ODBC driver version 1.0.3 and its documentation, see Connecting to Amazon Athena with ODBC (p. 185).

The streaming results feature is available with this new version of the ODBC driver. It is also available with the JDBC driver. For information about streaming results, see the ODBC Driver Installation and Configuration Guide, and search for UseResultsetStreaming.

The ODBC driver version 1.0.3 is a drop-in replacement for the previous version of the driver. We recommend that you migrate to the current driver.

Important
To use the ODBC driver version 1.0.3, follow these requirements:

- Keep the port 444 open to outbound traffic.
- Add the athena:GetQueryResultsStream policy action to the list of policies for Athena. This policy action is not exposed directly with the API and is only used with the ODBC and JDBC drivers, as part of streaming results support. For an example policy, see AWS managed policy: AWSQuicksightAthenaAccess (p. 534).

August 23, 2018

Published on 2018-08-23

Added support for these DDL-related features and fixed several bugs, as follows:

- Added support for BINARY and DATE data types for data in Parquet, and for DATE and TIMESTAMP data types for data in Avro.
- Added support for INT and DOUBLE in DDL queries. INTEGER is an alias to INT, and DOUBLE PRECISION is an alias to DOUBLE.
- Improved performance of DROP TABLE and DROP DATABASE queries.
- Removed the creation of _$folder$ object in Amazon S3 when a data bucket is empty.
• Fixed an issue where ALTER TABLE ADD PARTITION threw an error when no partition value was provided.
• Fixed an issue where DROP TABLE ignored the database name when checking partitions after the qualified name had been specified in the statement.

For more about the data types supported in Athena, see Data types in Amazon Athena (p. 690).

For information about supported data type mappings between types in Athena, the JDBC driver, and Java data types, see the "Data Types" section in the JDBC Driver Installation and Configuration Guide.

August 16, 2018

Published on 2018-08-16

Released the JDBC driver version 2.0.5. The new version of the JDBC driver streams results by default, instead of paging through them, allowing business intelligence tools to retrieve large data sets faster. Compared to the previous version of the JDBC driver, there are the following performance improvements:

• Approximately 2x performance increase when fetching less than 10K rows.
• Approximately 5-6x performance increase when fetching more than 10K rows.

The streaming results feature is available only with the JDBC driver. It is not available with the ODBC driver. You cannot use it with the Athena API. For information about streaming results, see the JDBC Driver Installation and Configuration Guide, and search for UseResultSetStreaming.

For downloading the JDBC driver version 2.0.5 and its documentation, see Using Athena with the JDBC driver (p. 183).

The JDBC driver version 2.0.5 is a drop-in replacement for the previous version of the driver (2.0.2). To ensure that you can use the JDBC driver version 2.0.5, add the athena:GetQueryResultsStream policy action to the list of policies for Athena. This policy action is not exposed directly with the API and is only used with the JDBC driver, as part of streaming results support. For an example policy, see AWS managed policy: AWSQuicksightAthenaAccess (p. 534). For more information about migrating from version 2.0.2 to version 2.0.5 of the driver, see the JDBC Driver Migration Guide.

If you are migrating from a 1.x driver to a 2.x driver, you will need to migrate your existing configurations to the new configuration. We highly recommend that you migrate to the current version of the driver. For more information, see the JDBC Driver Migration Guide.

August 7, 2018

Published on 2018-08-07

You can now store Amazon Virtual Private Cloud flow logs directly in Amazon S3 in a GZIP format, where you can query them in Athena. For information, see Querying Amazon VPC flow logs (p. 474) and Amazon VPC Flow Logs can now be delivered to S3.

June 5, 2018

Published on 2018-06-05

Topics
• Support for Views (p. 832)
• Improvements and Updates to Error Messages (p. 832)
• Support for Views

Support for Views

Added support for views. You can now use CREATE VIEW (p. 750), DESCRIBE VIEW (p. 754), DROP VIEW (p. 756), SHOW CREATE VIEW (p. 760), and SHOW VIEWS (p. 764) in Athena. The query that defines the view runs each time you reference the view in your query. For more information, see Working with views (p. 361).

• Improvements and Updates to Error Messages

Improvements and Updates to Error Messages

• Included a GSON 2.8.0 library into the CloudTrail SerDe, to solve an issue with the CloudTrail SerDe and enable parsing of JSON strings.
• Enhanced partition schema validation in Athena for Parquet, and, in some cases, for ORC, by allowing reordering of columns. This enables Athena to better deal with changes in schema evolution over time, and with tables added by the AWS Glue Crawler. For more information, see Handling schema updates (p. 376).
• Added parsing support for SHOW VIEWS.
• Made the following improvements to most common error messages:
  • Replaced an Internal Error message with a descriptive error message when a SerDe fails to parse the column in an Athena query. Previously, Athena issued an internal error in cases of parsing errors. The new error message reads: "HIVE_BAD_DATA: Error parsing field value for field 0: java.lang.String cannot be cast to org.openx.data.jsonserde.json.JSONObject".
  • Improved error messages about insufficient permissions by adding more detail.

• Bug Fixes

Bug Fixes

Fixed the following bugs:

• Fixed an issue that enables the internal translation of REAL to FLOAT data types. This improves integration with the AWS Glue crawler that returns FLOAT data types.
• Fixed an issue where Athena was not converting AVRO DECIMAL (a logical type) to a DECIMAL type.
• Fixed an issue where Athena did not return results for queries on Parquet data with WHERE clauses that referenced values in the TIMESTAMP data type.

May 17, 2018

Published on 2018-05-17

Increased query concurrency quota in Athena from five to twenty. This means that you can submit and run up to twenty DDL queries and twenty SELECT queries at a time. Note that the concurrency quotas are separate for DDL and SELECT queries.

Concurrency quotas in Athena are defined as the number of queries that can be submitted to the service concurrently. You can submit up to twenty queries of the same type (DDL or SELECT) at a time. If you submit a query that exceeds the concurrent query quota, the Athena API displays an error message.

After you submit your queries to Athena, it processes the queries by assigning resources based on the overall service load and the amount of incoming requests. We continuously monitor and make adjustments to the service so that your queries process as fast as possible.

For information, see Service Quotas (p. 785). This is an adjustable quota. You can use the Service Quotas console to request a quota increase for concurrent queries.
April 19, 2018
Published on 2018-04-19
Released the new version of the JDBC driver (version 2.0.2) with support for returning the ResultSet data as an Array data type, improvements, and bug fixes. For details, see the Release Notes for the driver.

For information about downloading the new JDBC driver version 2.0.2 and its documentation, see Using Athena with the JDBC driver (p. 183).

The latest version of the JDBC driver is 2.0.2. If you are migrating from a 1.x driver to a 2.x driver, you will need to migrate your existing configurations to the new configuration. We highly recommend that you migrate to the current driver.

For information about the changes introduced in the new version of the driver, the version differences, and examples, see the JDBC Driver Migration Guide.

April 6, 2018
Published on 2018-04-06
Use auto-complete to type queries in the Athena console.

March 15, 2018
Published on 2018-03-15
Added an ability to automatically create Athena tables for CloudTrail log files directly from the CloudTrail console. For information, see Using the CloudTrail console to create an Athena table for CloudTrail logs (p. 456).

February 2, 2018
Published on 2018-02-12
Added an ability to securely offload intermediate data to disk for memory-intensive queries that use the GROUP BY clause. This improves the reliability of such queries, preventing "Query resource exhausted" errors.

January 19, 2018
Published on 2018-01-19
Athena uses Presto, an open-source distributed query engine, to run queries.

With Athena, there are no versions to manage. We have transparently upgraded the underlying engine in Athena to a version based on Presto version 0.172. No action is required on your end.

With the upgrade, you can now use Presto 0.172 Functions and Operators, including Presto 0.172 Lambda Expressions in Athena.

Major updates for this release, including the community-contributed fixes, include:

- Support for ignoring headers. You can use the skip.header.line.count property when defining tables, to allow Athena to ignore headers. This is supported for queries that use the LazySimpleSerDe (p. 324) and OpenCSV SerDe (p. 329), and not for Grok or Regex SerDes.

- Support for the CHAR(n) data type in STRING functions. The range for CHAR(n) is [1, 255], while the range for VARCHAR(n) is [1, 65535].
• Support for correlated subqueries.
• Support for Presto Lambda expressions and functions.
• Improved performance of the DECIMAL type and operators.
• Support for filtered aggregations, such as `SELECT sum(col_name) FILTER, where id > 0`.
• Push-down predicates for the DECIMAL, TINYINT, SMALLINT, and REAL data types.
• Support for quantified comparison predicates: ALL, ANY, and SOME.
• Added functions: `arrays_overlap()`, `array_except()`, `levenshtein_distance()`, `codepoint()`, `skewness()`, `kurtosis()`, and `typeof()`.
• Added a variant of the `from_unixtime()` function that takes a timezone argument.
• Added the `bitwise_and_agg()` and `bitwise_or_agg()` aggregation functions.
• Added the `xxhash64()` and `to_big_endian_64()` functions.
• Added support for escaping double quotes or backslashes using a backslash with a JSON path subscript to the `json_extract()` and `json_extract_scalar()` functions. This changes the semantics of any invocation using a backslash, as backslashes were previously treated as normal characters.

For a complete list of functions and operators, see DML queries, functions, and operators (p. 692) in this guide, and Functions and operators in the Presto documentation.

Athena does not support all of Presto's features. For more information, see Limitations (p. 764).

Athena release notes for 2017

November 13, 2017

Published on 2017-11-13

Added support for connecting Athena to the ODBC Driver. For information, see Connecting to Amazon Athena with ODBC (p. 185).

November 1, 2017

Published on 2017-11-01

Added support for querying geospatial data, and for Asia Pacific (Seoul), Asia Pacific (Mumbai), and EU (London) regions. For information, see Querying geospatial data (p. 402) and AWS Regions and Endpoints.

October 19, 2017

Published on 2017-10-19

Added support for EU (Frankfurt). For a list of supported regions, see AWS Regions and Endpoints.

October 3, 2017

Published on 2017-10-03

Create named Athena queries with AWS CloudFormation. For more information, see `AWS::Athena::NamedQuery` in the AWS CloudFormation User Guide.
September 25, 2017
Published on 2017-09-25
Added support for Asia Pacific (Sydney). For a list of supported regions, see AWS Regions and Endpoints.

August 14, 2017
Published on 2017-08-14
Added integration with the AWS Glue Data Catalog and a migration wizard for updating from the Athena managed data catalog to the AWS Glue Data Catalog. For more information, see Integration with AWS Glue (p. 21).

August 4, 2017
Published on 2017-08-04
Added support for Grok SerDe, which provides easier pattern matching for records in unstructured text files such as logs. For more information, see Grok SerDe (p. 317). Added keyboard shortcuts to scroll through query history using the console (CTRL + ⇧/⇩ using Windows, CMD + ⇧/⇩ using Mac).

June 22, 2017
Published on 2017-06-22
Added support for Asia Pacific (Tokyo) and Asia Pacific (Singapore). For a list of supported regions, see AWS Regions and Endpoints.

June 8, 2017
Published on 2017-06-08
Added support for Europe (Ireland). For more information, see AWS Regions and Endpoints.

May 19, 2017
Published on 2017-05-19
Added an Amazon Athena API and AWS CLI support for Athena; updated JDBC driver to version 1.1.0; fixed various issues.

• Amazon Athena enables application programming for Athena. For more information, see Amazon Athena API Reference. The latest AWS SDKs include support for the Athena API. For links to documentation and downloads, see the SDKs section in Tools for Amazon Web Services.
• The AWS CLI includes new commands for Athena. For more information, see the Amazon Athena API Reference.
• A new JDBC driver 1.1.0 is available, which supports the new Athena API as well as the latest features and bug fixes. Download the driver at https://s3.amazonaws.com/athena-downloads/drivers/AthenaJDBC41-1.1.0.jar. We recommend upgrading to the latest Athena JDBC driver; however, you may still use the earlier driver version. Earlier driver versions do not support the Athena API. For more information, see Using Athena with the JDBC driver (p. 183).
• Actions specific to policy statements in earlier versions of Athena have been deprecated. If you upgrade to JDBC driver version 1.1.0 and have customer-managed or inline IAM policies attached to
JDBC users, you must update the IAM policies. In contrast, earlier versions of the JDBC driver do not support the Athena API, so you can specify only deprecated actions in policies attached to earlier version JDBC users. For this reason, you shouldn't need to update customer-managed or inline IAM policies.

- These policy-specific actions were used in Athena before the release of the Athena API. Use these deprecated actions in policies only with JDBC drivers earlier than version 1.1.0. If you are upgrading the JDBC driver, replace policy statements that allow or deny deprecated actions with the appropriate API actions as listed or errors will occur:

<table>
<thead>
<tr>
<th>Deprecated Policy-Specific Action</th>
<th>Corresponding Athena API Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>athena:RunQuery</td>
<td>athena:StartQueryExecution</td>
</tr>
<tr>
<td>athena:CancelQueryExecution</td>
<td>athena:StopQueryExecution</td>
</tr>
<tr>
<td>athena:GetQueryExecutions</td>
<td>athena:ListQueryExecutions</td>
</tr>
</tbody>
</table>

**Improvements**

- Increased the query string length limit to 256 KB.

**Bug Fixes**

- Fixed an issue that caused query results to look malformed when scrolling through results in the console.
- Fixed an issue where a \u0000 character string in Amazon S3 data files would cause errors.
- Fixed an issue that caused requests to cancel a query made through the JDBC driver to fail.
- Fixed an issue that caused the AWS CloudTrail SerDe to fail with Amazon S3 data in US East (Ohio).
- Fixed an issue that caused DROP TABLE to fail on a partitioned table.

**April 4, 2017**

Published on 2017-04-04

Added support for Amazon S3 data encryption and released JDBC driver update (version 1.0.1) with encryption support, improvements, and bug fixes.

**Features**

- Added the following encryption features:
  - Support for querying encrypted data in Amazon S3.
  - Support for encrypting Athena query results.
- A new version of the driver supports new encryption features, adds improvements, and fixes issues.
- Added the ability to add, replace, and change columns using ALTER TABLE. For more information, see Alter Column in the Hive documentation.
- Added support for querying LZO-compressed data.
For more information, see Encryption at rest (p. 521).

**Improvements**

- Better JDBC query performance with page-size improvements, returning 1,000 rows instead of 100.
- Added ability to cancel a query using the JDBC driver interface.
- Added ability to specify JDBC options in the JDBC connection URL. See Using Athena with the JDBC driver (p. 183) for the most current JDBC driver.
- Added PROXY setting in the driver, which can now be set using ClientConfiguration in the AWS SDK for Java.

**Bug Fixes**

Fixed the following bugs:

- Throttling errors would occur when multiple queries were issued using the JDBC driver interface.
- The JDBC driver would stop when projecting a decimal data type.
- The JDBC driver would return every data type as a string, regardless of how the data type was defined in the table. For example, selecting a column defined as an INT data type using resultSet.GetObject() would return a STRING data type instead of INT.
- The JDBC driver would verify credentials at the time a connection was made, rather than at the time a query would run.
- Queries made through the JDBC driver would fail when a schema was specified along with the URL.

**March 24, 2017**

Published on 2017-03-24

Added the AWS CloudTrail SerDe, improved performance, fixed partition issues.

**Features**

- Added the AWS CloudTrail SerDe, which has since been superseded by the Hive JSON SerDe (p. 320) for reading CloudTrail logs. For information about querying CloudTrail logs, see Querying AWS CloudTrail logs (p. 454).

**Improvements**

- Improved performance when scanning a large number of partitions.
- Improved performance on MSCK Repair Table operation.
- Added ability to query Amazon S3 data stored in regions other than your primary Region. Standard inter-region data transfer rates for Amazon S3 apply in addition to standard Athena charges.

**Bug Fixes**

- Fixed a bug where a "table not found error" might occur if no partitions are loaded.
- Fixed a bug to avoid throwing an exception with ALTER TABLE ADD PARTITION IF NOT EXISTS queries.
- Fixed a bug in DROP PARTITIONS.
February 20, 2017

Published on 2017-02-20

Added support for AvroSerDe and OpenCSVSerDe, US East (Ohio) Region, and bulk editing columns in the console wizard. Improved performance on large Parquet tables.

Features

- Introduced support for new SerDes:
  - Avro SerDe (p. 315)
  - OpenCSVSerDe for processing CSV (p. 329)
- US East (Ohio) Region (us-east-2) launch. You can now run queries in this region.
- You can now use the Create Table From S3 bucket data form to define table schema in bulk. In the query editor, choose Create, S3 bucket data, and then choose Bulk add columns in the Column details section.
Bulk add columns

Define columns in name value pairs, using commas to separate definitions (col1_name data_type, col2_name data_type, ...). Certain advanced data types (namely, structs) are not supported in this interface, but are supported using DDL statements.

id int, name string

Improvements

- Improved performance on large Parquet tables.
## Document history

**Latest documentation update: November 18, 2022.**

We update the documentation frequently to address your feedback. The following table describes important additions to the Amazon Athena documentation. Not all updates are represented.

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Release date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Added documentation for the Athena IBM Db2 connector.</td>
<td>You can use the Amazon Athena connector for IBM Db2 to query Db2 from Athena. For more information, see [Amazon Athena IBM Db2 connector](p. 93)</td>
<td>November 18, 2022</td>
</tr>
<tr>
<td>Added documentation for query result reuse.</td>
<td>When you re-run a query in Athena, you can now optionally choose to reuse the last stored query result. This can increase performance and reduce costs in terms of the number of bytes scanned. For more information, see [Reusing query results](p. 354).</td>
<td>November 8, 2022</td>
</tr>
<tr>
<td>Updated documentation for CloudTrail logs.</td>
<td>The CREATE TABLE DDL for querying CloudTrail logs has been updated to use the JSON SerDe instead of the CloudTrail SerDe. For more information, see [Querying AWS CloudTrail logs](p. 454).</td>
<td>November 3, 2022</td>
</tr>
<tr>
<td>Added documentation for Athena engine version 3.</td>
<td>For more information about Athena engine version 3, see [Athena engine version 3](p. 672).</td>
<td>October 13, 2022</td>
</tr>
<tr>
<td>Added tutorial on configuring SSO for ODBC using the Okta plugin.</td>
<td>Configure the Amazon Athena ODBC driver and the Okta plugin for single sign-on (SSO) capability using the Okta identity provider. For more information, see [Configuring SSO for ODBC using the Okta plugin and Okta Identity Provider](p. 186).</td>
<td>August 23, 2022</td>
</tr>
<tr>
<td>Added tutorial on configuring SSO for Azure AD single-account access using ODBC.</td>
<td>Configure Azure Active Directory and ODBC for single sign-on to Athena with and without MFA support. AWS single-account access enables you to federate Azure AD to a single AWS account and use Azure AD to manage access to AWS IAM roles. For each AWS account, Azure AD administrators federate to AWS IAM, assign users or groups to the account, and configure Azure AD to authorize role access. For more information, see [Configuring SSO for Azure AD Single-Account Access using ODBC](p. 213).</td>
<td>August 16, 2022</td>
</tr>
<tr>
<td>Added documentation for viewing query plans and statistics in the Athena console.</td>
<td>You can use the Athena query editor to see graphical representations of how your queries will be run and graphs, details, and statistics of how completed queries ran. For more information, see [Viewing execution plans for SQL queries](p. 340) and [Viewing statistics and execution details for completed queries](p. 357).</td>
<td>July 21, 2022</td>
</tr>
<tr>
<td>Added documentation for querying</td>
<td>You can use Athena to query Apache views created in external Hive metastores. Some Hive functions are</td>
<td>April 22, 2022</td>
</tr>
<tr>
<td>Change Description</td>
<td>Release date</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>Apache Hive views in external Hive metastores.</td>
<td>February 28, 2022</td>
<td></td>
</tr>
<tr>
<td>You can use the saved queries feature in Athena to save, recall, edit, and rename your queries. For more information, see Using saved queries (p. 368) in this guide and UpdateNamedQuery in the Amazon Athena API Reference.</td>
<td>November 30, 2021</td>
<td></td>
</tr>
<tr>
<td>You can use Athena to read Lake Formation governed tables. Use Lake Formation governed tables to secure your tables in Amazon S3 by granting permissions at the column, row, and cell levels. For more information, see Using governed tables (p. 498).</td>
<td>November 26, 2021</td>
<td></td>
</tr>
<tr>
<td>Athena supports read, time travel, and write queries for Apache Iceberg tables that use the Apache Parquet format for data and the AWS Glue catalog for their metastore. For more information, see Using Iceberg tables (p. 503).</td>
<td>November 12, 2021</td>
<td></td>
</tr>
<tr>
<td>You can use the cross-account federated query feature to query data sources in another account. For information about setting up permissions to enable this feature, see Enabling cross-account federated queries (p. 171).</td>
<td>August 5, 2021</td>
<td></td>
</tr>
<tr>
<td>Use the UNLOAD statement to write query the results from a SELECT statement to the Apache Parquet, ORC, Apache Avro, and JSON formats. For more information, see UNLOAD (p. 717).</td>
<td>April 5, 2021</td>
<td></td>
</tr>
<tr>
<td>For more information, see Using EXPLAIN and EXPLAIN ANALYZE in Athena (p. 703) and Understanding Athena EXPLAIN statement results (p. 712).</td>
<td>December 30, 2020</td>
<td></td>
</tr>
<tr>
<td>For more information, see Troubleshooting in Athena (p. 766) and Performance tuning in Athena (p. 781).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>For more information, see Athena engine versioning (p. 669).</td>
<td>November 11, 2020</td>
<td></td>
</tr>
<tr>
<td>For more information, see Using Amazon Athena Federated Query (p. 60) and Using Athena with CalledVia context keys (p. 555).</td>
<td>November 11, 2020</td>
<td></td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Release date</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Added documentation for using the JDBC driver with Lake Formation for federated access to Athena.</td>
<td>For more information, see Using Lake Formation and the Athena JDBC and ODBC drivers for federated access to Athena (p. 589) and Tutorial: Configuring federated access for Okta users to Athena using Lake Formation and JDBC (p. 590).</td>
<td>September 25, 2020</td>
</tr>
<tr>
<td>Added documentation for the Amazon Athena OpenSearch data connector.</td>
<td>For more information, see Amazon Athena OpenSearch connector (p. 118).</td>
<td>July 21, 2020</td>
</tr>
<tr>
<td>Added documentation for querying Hudi datasets.</td>
<td>For more information, see Using Athena to query Apache Hudi datasets (p. 420).</td>
<td>July 9, 2020</td>
</tr>
<tr>
<td>Added documentation on querying Apache web server logs and IIS web server logs stored in Amazon S3.</td>
<td>For more information, see Querying Apache logs stored in Amazon S3 (p. 490) and Querying internet information server (IIS) logs stored in Amazon S3 (p. 492).</td>
<td>July 8, 2020</td>
</tr>
<tr>
<td>Added documentation for the general release of the Athena Data Connector for External Hive Metastore.</td>
<td>For more information, see Using Athena Data Connector for External Hive Metastore (p. 35).</td>
<td>June 1, 2020</td>
</tr>
<tr>
<td>Added documentation for tagging data catalog resources.</td>
<td>For more information, see Tagging Athena resources (p. 658).</td>
<td>June 1, 2020</td>
</tr>
<tr>
<td>Added documentation on partition projection.</td>
<td>For more information, see Partition projection with Amazon Athena (p. 265).</td>
<td>May 21, 2020</td>
</tr>
<tr>
<td>Updated the Java code examples for Athena.</td>
<td>For more information, see Code samples (p. 788).</td>
<td>May 11, 2020</td>
</tr>
<tr>
<td>Added a topic on querying Amazon GuardDuty findings.</td>
<td>For more information, see Querying Amazon GuardDuty findings (p. 467).</td>
<td>March 19, 2020</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Release date</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Added a topic on using CloudWatch Events to monitor Athena query state transitions.</td>
<td>For more information, see Monitoring Athena queries with CloudWatch events (p. 651).</td>
<td>March 11, 2020</td>
</tr>
<tr>
<td>Added a topic on querying AWS Global Accelerator flow logs with Athena.</td>
<td>For more information, see Querying AWS Global Accelerator flow logs (p. 465).</td>
<td>February 6, 2020</td>
</tr>
<tr>
<td>• Added documentation on using CTAS with INSERT INTO to add data from an unpartitioned source to a partitioned destination.</td>
<td>Documentation updates include, but are not limited to, the following topics:</td>
<td>February 4, 2020</td>
</tr>
<tr>
<td>• Added download links for the 1.1.0 preview version of the ODBC driver for Athena.</td>
<td>• Using CTAS and INSERT INTO for ETL and data analysis (p. 290)</td>
<td></td>
</tr>
<tr>
<td>• Description for SHOW DATABASES LIKE regex corrected.</td>
<td>• Connecting to Amazon Athena with ODBC (p. 185) (The 1.1.0 preview features are now included in the 1.1.2 ODBC driver.)</td>
<td></td>
</tr>
<tr>
<td>• Corrected partitioned_by syntax in CTA topic.</td>
<td>• SHOW DATABASES (p. 760)</td>
<td></td>
</tr>
<tr>
<td>• Other minor fixes.</td>
<td>• CREATE TABLE AS (p. 744)</td>
<td></td>
</tr>
<tr>
<td>Added documentation on using CTAS with INSERT INTO to add data from a partitioned source to a partitioned destination.</td>
<td>For more information, see Using CTAS and INSERT INTO to create a table with more than 100 partitions (p. 295).</td>
<td>January 22, 2020</td>
</tr>
<tr>
<td>Query results location information updated.</td>
<td>Athena no longer creates a 'default' query results location. For more information, see Specifying a query result location (p. 344).</td>
<td>January 20, 2020</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Release date</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>--------------</td>
</tr>
</tbody>
</table>
| Added topic on querying the AWS Glue Data Catalog. Updated information about service quotas (formerly “service limits”) in Athena. | For more information, see the following topics:  
  - Querying AWS Glue Data Catalog (p. 442)  
  - Service Quotas (p. 785) | January 17, 2020 |
| Corrected topic on OpenCSVSerDe to note that the TIMESTAMP type should be specified in the UNIX numeric format. | For more information, see OpenCSVSerDe for processing CSV (p. 329). | January 15, 2020 |
| Updated security topic on encryption to note that Athena does not support asymmetric keys. | Athena supports only symmetric keys for reading and writing data.  
For more information, see Supported Amazon S3 encryption options (p. 521). | January 8, 2020 |
| Added information about cross-account access to Amazon S3 buckets that are encrypted with a custom AWS KMS key. | For more information, see Cross-account access to a bucket encrypted with a custom AWS KMS key (p. 538). | December 13, 2019 |
| Added documentation for federated queries, external Hive metastores, machine learning, and user defined functions. Added new CloudWatch metrics. | For more information, see the following topics:  
  - Using Amazon Athena Federated Query (p. 60)  
  - Using Athena data source connectors (p. 67)  
  - Using Athena Data Connector for External Hive Metastore (p. 35)  
  - Using Machine Learning (ML) with Amazon Athena (p. 432)  
  - Querying with user defined functions (p. 434)  
  - List of CloudWatch metrics and dimensions for Athena (p. 650) | November 26, 2019 |
<p>| Added section for new INSERT INTO command and updated query result location information for supporting data manifest files. | For more information, see INSERT INTO (p. 697) and Working with query results, recent queries, and output files (p. 344). | September 18, 2019 |</p>
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Release date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Added section for interface VPC endpoints (PrivateLink) support. Updated JDBC drivers. Updated information about enriched VPC flow logs.</td>
<td>For more information, see Connect to Amazon Athena using an interface VPC endpoint (p. 579), Querying Amazon VPC flow logs (p. 474), and Using Athena with the JDBC driver (p. 183).</td>
<td>September 11, 2019</td>
</tr>
<tr>
<td>Added section on integrating with AWS Lake Formation.</td>
<td>For more information, see Using Athena to query data registered with AWS Lake Formation (p. 580).</td>
<td>June 26, 2019</td>
</tr>
<tr>
<td>Updated Security section for consistency with other AWS services.</td>
<td>For more information, see Amazon Athena security (p. 519).</td>
<td>June 26, 2019</td>
</tr>
<tr>
<td>Added section on querying AWS WAF logs.</td>
<td>For more information, see Querying AWS WAF logs (p. 481).</td>
<td>May 31, 2019</td>
</tr>
<tr>
<td>Released the new version of the ODBC driver with support for Athena workgroups.</td>
<td>To download the ODBC driver version 1.0.5 and its documentation, see Connecting to Amazon Athena with ODBC (p. 185). There are no changes to the ODBC driver connection string when you use tags on workgroups. To use tags, upgrade to the latest version of the ODBC driver, which is this current version. This driver version lets you use Athena API workgroup actions (p. 645) to create and manage workgroups, and Athena API tag actions (p. 660) to add, list, or remove tags on workgroups. Before you begin, make sure that you have resource-level permissions in IAM for actions on workgroups and tags.</td>
<td>March 5, 2019</td>
</tr>
<tr>
<td>Added tag support for workgroups in Amazon Athena.</td>
<td>A tag consists of a key and a value, both of which you define. When you tag a workgroup, you assign custom metadata to it. For example, create a workgroup for each cost center. Then, by adding tags to these workgroups, you can track your Athena spending for each cost center. For more information, see Using tags for billing in the AWS Billing and Cost Management User Guide.</td>
<td>February 22, 2019</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Release date</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>--------------</td>
</tr>
</tbody>
</table>
| Improved the JSON OpenX SerDe used in Athena. | The improvements include, but are not limited to, the following:  
- Support for the `ConvertDotsInJsonKeysToUnderscores` property. When set to TRUE, it allows the SerDe to replace the dots in key names with underscores. For example, if the JSON dataset contains a key with the name "a.b", you can use this property to define the column name to be "a_b" in Athena. The default is FALSE. By default, Athena does not allow dots in column names.  
- Support for the `case.insensitive` property. By default, Athena requires that all keys in your JSON dataset use lowercase. Using `WITH SERDE PROPERTIES ("case.insensitive"= FALSE;)` allows you to use case-sensitive key names in your data. The default is TRUE. When set to TRUE, the SerDe converts all uppercase columns to lowercase.  
For more information, see [OpenX JSON SerDe](p. 321). | February 18, 2019 |
<p>| Added support for workgroups. | Use workgroups to separate users, teams, applications, or workloads, and to set limits on the amount of data each query or the entire workgroup can process. Because workgroups act as IAM resources, you can use resource-level permissions to control access to a specific workgroup. You can also view query-related metrics in Amazon CloudWatch, control query costs by configuring limits on the amount of data scanned, create thresholds, and trigger actions, such as Amazon SNS alarms, when these thresholds are breached. For more information, see [Using workgroups for running queries](p. 629) and [Controlling costs and monitoring queries with CloudWatch metrics and events](p. 647). | February 18, 2019 |
| Added support for analyzing logs from Network Load Balancer. | Added example Athena queries for analyzing logs from Network Load Balancer. These logs receive detailed information about the Transport Layer Security (TLS) requests sent to the Network Load Balancer. You can use these access logs to analyze traffic patterns and troubleshoot issues. For information, see the section called “Network Load Balancer” (p. 470). | January 24, 2019 |
| Released the new versions of the JDBC and ODBC driver with support for federated access to Athena API with the AD FS and SAML 2.0 (Security Assertion Markup Language 2.0). | With this release of the drivers, federated access to Athena is supported for the Active Directory Federation Service (AD FS 3.0). Access is established through the versions of JDBC or ODBC drivers that support SAML 2.0. For information about configuring federated access to the Athena API, see the section called “Enabling federated access to the Athena API” (p. 571). | November 10, 2018 |</p>
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Release date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Added support for fine-grained</td>
<td>Added support for creating identity-based (IAM) policies that provide fine-grained access control to resources in the AWS Glue Data Catalog, such as databases and tables used in Athena.</td>
<td>October 15, 2018</td>
</tr>
<tr>
<td>access control to databases and</td>
<td>Additionally, you can encrypt database and table metadata in the Data Catalog, by adding specific policies to Athena.</td>
<td></td>
</tr>
<tr>
<td>tables in Athena.</td>
<td>For details, see <a href="#">Fine-grained access to databases and tables in the AWS Glue Data Catalog</a>.</td>
<td></td>
</tr>
<tr>
<td>Additionally, added policies in</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Athena that allow you to encrypt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>database and table metadata in the</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Catalog.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Added support for CREATE TABLE AS</td>
<td>Added support for CREATE TABLE AS SELECT statements. See <a href="#">Creating a table from query results (CTAS)</a>, <a href="#">Considerations and limitations for CTAS queries</a>, and <a href="#">Examples of CTAS queries</a>.</td>
<td>October 10, 2018</td>
</tr>
<tr>
<td>SELECT statements.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Made other improvements in the</td>
<td></td>
<td></td>
</tr>
<tr>
<td>documentation.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Released the ODBC driver version</td>
<td>The ODBC driver version 1.0.3 supports streaming results and also includes improvements, bug fixes, and an updated documentation for &quot;Using SSL with a Proxy Server&quot;.</td>
<td>September 6, 2018</td>
</tr>
<tr>
<td>1.0.3 with support for streaming</td>
<td>For downloading the ODBC driver version 1.0.3 and its documentation, see <a href="#">Connecting to Amazon Athena with ODBC</a>.</td>
<td></td>
</tr>
<tr>
<td>results instead of fetching them in</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pages.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Made other improvements in the</td>
<td></td>
<td></td>
</tr>
<tr>
<td>documentation.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Released the JDBC driver version</td>
<td>Released the JDBC driver 2.0.5 with default support for streaming results instead of fetching them in pages. For information, see <a href="#">Using Athena with the JDBC driver</a>.</td>
<td>August 16, 2018</td>
</tr>
<tr>
<td>2.0.5 with default support for</td>
<td></td>
<td></td>
</tr>
<tr>
<td>streaming results instead of</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fetching them in pages.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Made other improvements in the</td>
<td></td>
<td></td>
</tr>
<tr>
<td>documentation.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Updated the documentation for</td>
<td>Updated the documentation for querying Amazon Virtual Private Cloud flow logs, which can be stored directly in Amazon S3 in a GZIP format. For information, see <a href="#">Querying Amazon VPC flow logs</a>.</td>
<td>August 7, 2018</td>
</tr>
<tr>
<td>querying Amazon Virtual Private</td>
<td>Updated examples for querying ALB logs. For information, see <a href="#">Querying Application Load Balancer logs</a>.</td>
<td></td>
</tr>
<tr>
<td>Cloud flow logs, which can be</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stored directly in Amazon S3 in a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GZIP format.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Updated examples for querying ALB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>logs.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Release date</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Added support for views. Added guidelines for schema manipulations for various data storage formats.</td>
<td>Added support for views. For information, see Working with views (p. 361). Updated this guide with guidance on handling schema updates for various data storage formats. For information, see Handling schema updates (p. 376).</td>
<td>June 5, 2018</td>
</tr>
<tr>
<td>Increased default query concurrency limits from five to twenty.</td>
<td>You can submit and run up to twenty DDL queries and twenty SELECT queries at a time. For information, see Service Quotas (p. 785).</td>
<td>May 17, 2018</td>
</tr>
<tr>
<td>Added query tabs, and an ability to configure auto-complete in the Query Editor.</td>
<td>Added query tabs, and an ability to configure auto-complete in the Query Editor. For information, see Using the console (p. 19).</td>
<td>May 8, 2018</td>
</tr>
<tr>
<td>Released the JDBC driver version 2.0.2.</td>
<td>Released the new version of the JDBC driver (version 2.0.2). For information, see Using Athena with the JDBC driver (p. 183).</td>
<td>April 19, 2018</td>
</tr>
<tr>
<td>Added auto-complete for typing queries in the Athena console.</td>
<td>Added auto-complete for typing queries in the Athena console.</td>
<td>April 6, 2018</td>
</tr>
<tr>
<td>Added an ability to create Athena tables for CloudTrail log files directly from the CloudTrail console.</td>
<td>Added an ability to automatically create Athena tables for CloudTrail log files directly from the CloudTrail console. For information, see Using the CloudTrail console to create an Athena table for CloudTrail logs (p. 456).</td>
<td>March 15, 2018</td>
</tr>
<tr>
<td>Added support for securely offloading intermediate data to disk for queries with GROUP BY.</td>
<td>Added an ability to securely offload intermediate data to disk for memory-intensive queries that use the GROUP BY clause. This improves the reliability of such queries, preventing &quot;Query resource exhausted&quot; errors. For more information, see the release note for February 2, 2018 (p. 833).</td>
<td>February 2, 2018</td>
</tr>
<tr>
<td>Added support for Presto version 0.172.</td>
<td>upgraded the underlying engine in Amazon Athena to a version based on Presto version 0.172. For more information, see the release note for January 19, 2018 (p. 833).</td>
<td>January 19, 2018</td>
</tr>
<tr>
<td>Added support for the ODBC Driver.</td>
<td>Added support for connecting Athena to the ODBC Driver. For information, see Connecting to Amazon Athena with ODBC.</td>
<td>November 13, 2017</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Release date</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Added support for Asia Pacific (Seoul), Asia Pacific (Mumbai), and Europe (London) regions. Added support for querying geospatial data.</td>
<td>Added support for querying geospatial data, and for Asia Pacific (Seoul), Asia Pacific (Mumbai), Europe (London) regions. For information, see Querying geospatial data and AWS Regions and endpoints.</td>
<td>November 1, 2017</td>
</tr>
<tr>
<td>Added support for Europe (Frankfurt).</td>
<td>Added support for Europe (Frankfurt). For a list of supported regions, see AWS Regions and endpoints.</td>
<td>October 19, 2017</td>
</tr>
<tr>
<td>Added support for named Athena queries with AWS CloudFormation.</td>
<td>Added support for creating named Athena queries with AWS CloudFormation. For more information, see AWS::Athena::NamedQuery in the AWS CloudFormation User Guide.</td>
<td>October 3, 2017</td>
</tr>
<tr>
<td>Added support for Asia Pacific (Sydney).</td>
<td>Added support for Asia Pacific (Sydney). For a list of supported regions, see AWS Regions and endpoints.</td>
<td>September 25, 2017</td>
</tr>
<tr>
<td>Added a section to this guide for querying AWS service logs and different types of data, including maps, arrays, nested data, and data containing JSON.</td>
<td>Added examples for Querying AWS service logs (p. 447) and for querying different types of data in Athena. For information, see Running SQL queries using Amazon Athena (p. 339).</td>
<td>September 5, 2017</td>
</tr>
<tr>
<td>Added support for AWS Glue Data Catalog.</td>
<td>Added integration with the AWS Glue Data Catalog and a migration wizard for updating from the Athena managed data catalog to the AWS Glue Data Catalog. For more information, see Integration with AWS Glue and AWS Glue.</td>
<td>August 14, 2017</td>
</tr>
<tr>
<td>Added support for Grok SerDe.</td>
<td>Added support for Grok SerDe, which provides easier pattern matching for records in unstructured text files such as logs. For more information, see Grok SerDe. Added keyboard shortcuts to scroll through query history using the console.</td>
<td>August 4, 2017</td>
</tr>
<tr>
<td>Added support for Asia Pacific (Tokyo).</td>
<td>Added support for Asia Pacific (Tokyo) and Asia Pacific (Singapore). For a list of supported regions, see AWS Regions and endpoints.</td>
<td>June 22, 2017</td>
</tr>
<tr>
<td>Added support for Europe (Ireland).</td>
<td>Added support for Europe (Ireland). For more information, see AWS Regions and endpoints.</td>
<td>June 8, 2017</td>
</tr>
<tr>
<td>Added an Amazon Athena API and AWS CLI support.</td>
<td>Added an Amazon Athena API and AWS CLI support for Athena. Updated JDBC driver to version 1.1.0.</td>
<td>May 19, 2017</td>
</tr>
<tr>
<td>Added support for Amazon S3 data encryption.</td>
<td>Added support for Amazon S3 data encryption and released a JDBC driver update (version 1.0.1) with encryption support, improvements, and bug fixes. For more information, see Encryption at rest (p. 521).</td>
<td>April 4, 2017</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Release date</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>--------------</td>
</tr>
</tbody>
</table>
| Added the AWS CloudTrail SerDe. | Added the AWS CloudTrail SerDe, improved performance, fixed partition issues.  
- The AWS CloudTrail SerDe has been superseded by the Hive JSON SerDe (p. 320) for reading CloudTrail logs. For information about querying CloudTrail logs, see Querying AWS CloudTrail logs (p. 454).  
- Improved performance when scanning a large number of partitions.  
- Improved performance on MSCK Repair Table operation.  
- Added ability to query Amazon S3 data stored in regions other than your primary region. Standard inter-region data transfer rates for Amazon S3 apply in addition to standard Athena charges. | March 24, 2017 |
| Added support for US East (Ohio). | Added support for Avro SerDe (p. 315) and OpenCSVSerDe for processing CSV (p. 329), US East (Ohio), and bulk editing columns in the console wizard. Improved performance on large Parquet tables. | February 20, 2017 |
| | The initial release of the Amazon Athena User Guide. | November, 2016 |
AWS glossary

For the latest AWS terminology, see the AWS glossary in the AWS General Reference.