Amazon Athena

User Guide
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
Accessing Athena .................................................................................................................................................... 1
Understanding Tables, Databases, and the Data Catalog .................................................................................. 2
  AWS Service Integrations with Athena ........................................................................................................... 3
Setting Up ........................................................................................................................................................ 6
  1. Sign Up for an AWS Account .................................................................................................................. 6
  2. Create an IAM Administrator User and Group .................................................................................... 6
  3. Attach Managed Policies for Athena ........................................................................................................ 6
  4. Sign in as an IAM User ............................................................................................................................ 7
Getting Started ................................................................................................................................................. 8
  Prerequisites .................................................................................................................................................. 8
  Step 1: Create a Database ........................................................................................................................... 8
  Step 2: Create a Table .................................................................................................................................. 11
  Step 3: Query Data ...................................................................................................................................... 14
Connecting to Other Data Sources .................................................................................................................... 16
Accessing Amazon Athena .................................................................................................................................. 17
  Using the Console ........................................................................................................................................ 17
  Using the API ............................................................................................................................................... 17
  Using the CLI ................................................................................................................................................ 17
Connecting to Data Sources ................................................................................................................................... 18
  Integration with AWS Glue ........................................................................................................................ 18
    Using AWS Glue to Connect to Data Sources in Amazon S3 .................................................................. 19
    Registering an AWS Glue Data Catalog from Another Account ............................................................ 22
    Best Practices When Using Athena with AWS Glue ................................................................................ 23
    Upgrading to the AWS Glue Data Catalog Step-by-Step ....................................................................... 31
    FAQ: Upgrading to the AWS Glue Data Catalog ................................................................................. 34
  Using a Hive Metastore .................................................................................................................................. 36
    Overview of Features ................................................................................................................................. 37
    Workflow .................................................................................................................................................... 37
    Considerations and Limitations ................................................................................................................ 38
    Connecting Athena to an Apache Hive Metastore .................................................................................. 39
    Using the AWS Serverless Application Repository ............................................................................... 46
    Connecting Athena to Hive Using an Existing Role ........................................................................... 48
    Using a Default Catalog ............................................................................................................................ 58
    Using the AWS CLI with Hive Metastores .............................................................................................. 61
    Reference Implementation ........................................................................................................................ 67
  Using Amazon Athena Federated Query ......................................................................................................... 68
    Considerations and Limitations ................................................................................................................ 69
    Videos ........................................................................................................................................................ 69
    Deploying a Connector and Connecting to a Data Source .................................................................... 70
    Using the AWS Serverless Application Repository ............................................................................... 72
    Athena Data Source Connectors ............................................................................................................. 72
    Writing Federated Queries ...................................................................................................................... 76
    Writing a Data Source Connector ........................................................................................................... 79
IAM Policies for Accessing Data Catalogs ........................................................................................................... 80
  Data Catalog Example Policies .................................................................................................................... 81
Managing Data Sources ...................................................................................................................................... 84
Connecting to Amazon Athena with ODBC and JDBC Drivers ........................................................................ 86
  Using Athena with the JDBC Driver .......................................................................................................... 87
  Connecting to Amazon Athena with ODBC ............................................................................................. 88
  Power BI Connector ................................................................................................................................... 90
Creating Databases and Tables ......................................................................................................................... 97
Creating Databases .......................................................................................................................................... 97
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating Tables</td>
<td>99</td>
</tr>
<tr>
<td>Considerations and Limitations</td>
<td>100</td>
</tr>
<tr>
<td>Creating Tables Using AWS Glue or the Athena Console</td>
<td>101</td>
</tr>
<tr>
<td>Showing Table Information</td>
<td>105</td>
</tr>
<tr>
<td>Names for Tables, Databases, and Columns</td>
<td>106</td>
</tr>
<tr>
<td>Table names and table column names in Athena must be lowercase</td>
<td>106</td>
</tr>
<tr>
<td>Special characters</td>
<td>106</td>
</tr>
<tr>
<td>Reserved Keywords</td>
<td>107</td>
</tr>
<tr>
<td>List of Reserved Keywords in DDL Statements</td>
<td>107</td>
</tr>
<tr>
<td>List of Reserved Keywords in SQL SELECT Statements</td>
<td>107</td>
</tr>
<tr>
<td>Examples of Queries with Reserved Words</td>
<td>108</td>
</tr>
<tr>
<td>Table Location in Amazon S3</td>
<td>109</td>
</tr>
<tr>
<td>Table Location and Partitions</td>
<td>110</td>
</tr>
<tr>
<td>Converting to Columnar Formats</td>
<td>111</td>
</tr>
<tr>
<td>Overview</td>
<td>111</td>
</tr>
<tr>
<td>Before you begin</td>
<td>8</td>
</tr>
<tr>
<td>Example: Converting data to Parquet using an EMR cluster</td>
<td>113</td>
</tr>
<tr>
<td>Partitioning Data</td>
<td>114</td>
</tr>
<tr>
<td>Considerations and Limitations</td>
<td>115</td>
</tr>
<tr>
<td>Creating and Loading a Table with Partitioned Data</td>
<td>115</td>
</tr>
<tr>
<td>Preparing Partitioned and Nonpartitioned Data for Querying</td>
<td>116</td>
</tr>
<tr>
<td>Partition Projection</td>
<td>119</td>
</tr>
<tr>
<td>Pruning and Projection for Heavily Partitioned Tables</td>
<td>119</td>
</tr>
<tr>
<td>Using Partition Projection</td>
<td>119</td>
</tr>
<tr>
<td>Use Cases</td>
<td>120</td>
</tr>
<tr>
<td>Considerations and Limitations</td>
<td>120</td>
</tr>
<tr>
<td>Video</td>
<td>121</td>
</tr>
<tr>
<td>Setting up Partition Projection</td>
<td>121</td>
</tr>
<tr>
<td>Supported Types for Partition Projection</td>
<td>126</td>
</tr>
<tr>
<td>Dynamic ID Partitioning</td>
<td>130</td>
</tr>
<tr>
<td>Amazon Kinesis Data Firehose Example</td>
<td>132</td>
</tr>
<tr>
<td>SerDe Reference</td>
<td>133</td>
</tr>
<tr>
<td>Using a SerDe</td>
<td>133</td>
</tr>
<tr>
<td>To Use a SerDe in Queries</td>
<td>133</td>
</tr>
<tr>
<td>Supported SerDes and Data Formats</td>
<td>134</td>
</tr>
<tr>
<td>Avro SerDe</td>
<td>136</td>
</tr>
<tr>
<td>Regex SerDe</td>
<td>138</td>
</tr>
<tr>
<td>CloudTrail SerDe</td>
<td>138</td>
</tr>
<tr>
<td>OpenCSVSerDe for Processing CSV</td>
<td>140</td>
</tr>
<tr>
<td>Grok SerDe</td>
<td>143</td>
</tr>
<tr>
<td>JSON SerDe Libraries</td>
<td>146</td>
</tr>
<tr>
<td>LazySimpleSerDe for CSV, TSV, and Custom-Delimited Files</td>
<td>150</td>
</tr>
<tr>
<td>ORC SerDe</td>
<td>155</td>
</tr>
<tr>
<td>Parquet SerDe</td>
<td>157</td>
</tr>
<tr>
<td>Compression Formats</td>
<td>160</td>
</tr>
<tr>
<td>Specifying Compression Formats</td>
<td>160</td>
</tr>
<tr>
<td>Notes and Resources</td>
<td>160</td>
</tr>
<tr>
<td>Running Queries</td>
<td>162</td>
</tr>
<tr>
<td>Query Results and Query History</td>
<td>162</td>
</tr>
<tr>
<td>Getting a Query ID</td>
<td>163</td>
</tr>
<tr>
<td>Finding Query Output Files in Amazon S3</td>
<td>164</td>
</tr>
<tr>
<td>Downloading Query Results Files Using the Athena Console</td>
<td>166</td>
</tr>
<tr>
<td>Specifying a Query Result Location</td>
<td>167</td>
</tr>
<tr>
<td>Viewing Query History</td>
<td>170</td>
</tr>
<tr>
<td>Working with Views</td>
<td>172</td>
</tr>
<tr>
<td>When to Use Views?</td>
<td>172</td>
</tr>
</tbody>
</table>
Troubleshooting ............................................................................................................................. 530
Output Issues ......................................................................................................................... 533
MSCK REPAIR TABLE ............................................................................................................... 533
JSON Related Errors ............................................................................................................... 532
Federated Queries .................................................................................................................. 532
CREATE TABLE AS SELECT (CTAS) ............................................................................................. 530
Considerations and Limitations ................................................................................................. 528
The S3 location provided to save your query results is invalid ............................................... 532
Store Athena query output in a format other than CSV ....................................................... 533
TIMESTAMP result is empty ............................................................................................. 533
Unable to verify/create output bucket .............................................................................. 533
Multiple JSON records return a SELECT COUNT of 1 ........................................................... 533
HIVE_CURSOR_ERROR: Row is not a valid JSON Object - JSONException: Duplicate key ............ 533
org.openx.data.jsonserde.json.JSONObject ......................................................................... 532
NULL or incorrect data errors when trying to read JSON data ............................................... 532
The S3 location provided to save your query results is invalid. .............................................. 532
Athena reads files that I excluded from the AWS Glue crawler .............................................. 531
Athena cannot read hidden files ....................................................................................... 530
Athena cannot read files stored in the Glacier storage class ................................................. 530
Duplicated data occurs with concurrent CTAS statements .................................................... 530
Data File Issues ..................................................................................................................... 530
HIVE_BAD_DATA: Error parsing field value ......................................................................... 531
HIVE_CURSOR_ERROR: Unexpected end of input stream ...................................................... 531
HIVE_CURSOR_ERROR: com.amazonaws.services.s3.model.AmazonS3Exception: The specified key does not exist .......................................................... 531
HIVE_CANNOT_OPEN_SPLIT: Error opening Hive split s3://bucket-name .............................. 531
HIVE_UNKNOWN_ERROR: Unable to create input format ...................................................... 531
org.apache.parquet.io.GroupColumnIO cannot be cast to org.apache.parquet.io.HiveColumnIO ................................. 532
The S3 location provided to save your query results is invalid. .............................................. 532
Federated Queries .................................................................................................................. 532
JSON Related Errors ............................................................................................................... 532
NULL or incorrect data errors when trying to read JSON data ............................................... 532
HIVE_BAD_DATA: Error parsing field value for field 0: java.lang.String cannot be cast to org.openx.data.jsonserde.json.JSONArray .............................. 532
HIVE_CURSOR_ERROR: Row is not a valid JSON Object - JSONException: Duplicate key ............ 533
Multiple JSON records return a SELECT COUNT of 1 ........................................................... 533
Cannot query a table created by a AWS Glue crawler that uses a custom JSON classifier ...... 533
Output Issues ......................................................................................................................... 533
Unable to verify/create output bucket ................................................................................... 533
TIMESTAMP result is empty ............................................................................................. 533
Store Athena query output in a format other than CSV ....................................................... 533
The S3 location provided to save your query results is invalid .............................................. 534
Partitioning Issues ................................................................................................................................. 534
MSCK REPAIR TABLE does not remove stale partitions ........................................................................ 534
MSCK REPAIR TABLE failure .................................................................................................................. 534
MSCK REPAIR TABLE detects partitions but doesn’t add them to AWS Glue ........................................ 534
Partition projection ranges with the date format of dd-MM-yyyy-HH-mm-ss or yyyy-MM-dd do not work ................................................................................................................................. 534
PARTITION BY doesn’t support the BIGINT type .................................................................................... 534
No meaningful partitions available .......................................................................................................... 534
Partition projection does not work in conjunction with range partitions .................................................. 535
HIVE_UNKNOWN_ERROR: Unable to create input format ........................................................................ 535
HIVE_PARTITION_SCHEMA_MISMATCH .................................................................................................. 535
SemanticException table is not partitioned but partition spec exists ....................................................... 535
Zero records returned from partitioned data ............................................................................................ 535
Permissions ................................................................................................................................................ 535
Access Denied Error when querying Amazon S3 .................................................................................... 535
Access Denied with Status Code: 403 when running DDL queries on encrypted data in Amazon S3 .............................................................................................................................................. 535
Access Denied with Status Code: 403 when querying an Amazon S3 bucket in another account .......... 536
Use IAM role credentials to connect to the Athena JDBC driver ............................................................ 536
Query Syntax Issues .................................................................................................................................. 537
FAILED: NullPointerException Name is null .......................................................................................... 537
Function not registered .............................................................................................................................. 537
Number of matching groups doesn’t match the number of columns ...................................................... 537
queryString failed to satisfy constraint: Member must have length less than or equal to 262144 .......... 537
SYNTAX_ERROR: Column cannot be resolved .......................................................................................... 537
Throttling Issues ....................................................................................................................................... 538
Views .......................................................................................................................................................... 538
Views created in Apache Hive shell do not work in Athena ..................................................................... 538
View is stale; it must be re-created ........................................................................................................... 538
Workgroups .............................................................................................................................................. 538
Additional Resources ............................................................................................................................... 538
Performance Tuning .................................................................................................................................. 540
Physical Limits .......................................................................................................................................... 540
Query Optimization Techniques .................................................................................................................. 540
Data Size .................................................................................................................................................. 540
File Formats .............................................................................................................................................. 541
Joins, Grouping, and Unions ...................................................................................................................... 541
Partitioning ................................................................................................................................................ 542
Window Functions ........................................................................................................................................ 542
Use More Efficient Functions .................................................................................................................... 542
Additional Resources ................................................................................................................................. 543
Code Samples, Service Quotas, and Previous JDBC Driver ...................................................................... 544
Code Samples ............................................................................................................................................. 544
Constants .................................................................................................................................................. 544
Create a Client to Access Athena ............................................................................................................... 545
Start Query Execution ............................................................................................................................... 545
Stop Query Execution ............................................................................................................................... 547
List Query Executions ............................................................................................................................... 549
Create a Named Query .............................................................................................................................. 550
Delete a Named Query ............................................................................................................................... 551
List Named Queries .................................................................................................................................. 552
 Earlier Version JDBC Drivers ..................................................................................................................... 553
Instructions for JDBC Driver version 1.1.0 ................................................................................................. 554
Service Quotas .......................................................................................................................................... 558
Queries ....................................................................................................................................................... 558
Workgroups ................................................................................................................................................. 559
Databases, Tables, and Partitions ................................................................................................................. 559
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. 472). 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.

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. 8).
• To learn how to use JDBC or ODBC drivers, see Connecting to Amazon Athena with JDBC (p. 87) and Connecting to Amazon Athena with ODBC (p. 88).
• 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. 18). 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. 474) statements to query it, including getting specific file locations for your source data (p. 477). Your query results are stored in Amazon S3 in the query result location that you specify (p. 167).

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. 8).

• 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. 420) in the Amazon Athena User Guide and CreateWorkGroup in the Amazon Athena API Reference.

Amazon CloudFront

Reference topic: Querying Amazon CloudFront Logs (p. 274)

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

AWS CloudTrail

Reference topic: Querying AWS CloudTrail Logs (p. 276)

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 Creating a Table for CloudTrail Logs in the CloudTrail Console (p. 277).

Elastic Load Balancing

Reference topic: Querying Application Load Balancer Logs (p. 271)

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. 271).

Reference topic: Querying Classic Load Balancer Logs (p. 273)

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. 273).

AWS Glue Data Catalog

Reference topic: Integration with AWS Glue (p. 18)

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. 18) 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. 324).
Amazon QuickSight

Reference topic: Connecting to Amazon Athena with ODBC and JDBC Drivers (p. 86)

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. 86).

Amazon S3 Inventory

Reference topic: Querying inventory with Athena in the Amazon Simple Storage Service Developer 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 Developer 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. 294)

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. 6)
2. Create an IAM Administrator User and Group (p. 6)
3. Attach Managed Policies for Athena (p. 6)
4. Sign in as an IAM User (p. 7)

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. 325).
For steps to attach the Athena managed policies, follow Adding IAM Identity Permissions (Console) in the IAM User Guide and add the AmazonAthenaFullAccess and AWSQuicksightAthenaAccess 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 Amazon S3 Permissions (p. 331). If the dataset or Athena query results are encrypted, you may need additional permissions. For more information, see Configuring Encryption Options (p. 318).

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. 6).
- 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 a database

1. Open the Athena console.
2. If this is your first time visiting the Athena console in your current AWS Region, choose Get Started to open the Query Editor. If it isn't your first time, the Athena Query Editor opens.
3. Choose the link to set up a query result location in Amazon S3.
4. In the Settings dialog box, do one of the following:
   - Choose the Select folder icon, choose the Amazon S3 bucket that you created in your current Region, and then choose Select.
   - In the Query result location text box, enter the path to the bucket that you created in Amazon S3 for your query results. Prefix the path with s3:// and add a forward slash to the end of the path.
5. Choose **Save**.

If you ever need to return to the **Settings** dialog box, choose the **Settings** option at the top of the Athena console.

6. On the right of the navigation pane, you can use the Athena Query Editor to enter and run queries and statements.
7. To create a database named `mydatabase`, enter the following CREATE DATABASE statement.

```sql
CREATE DATABASE mydatabase
```

8. Choose Run Query or press `Ctrl+ENTER`.

9. In the navigation pane on the left, from the Database list, choose `mydatabase` to make it your current database.
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.

```
CREATE EXTERNAL TABLE IF NOT EXISTS cloudfront_logs (  'Date' DATE,  Time STRING,  Location STRING,  Bytes INT,  RequestIP STRING,  Method STRING,  Host STRING,  Uri STRING,```
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. 107), it is escaped with backtick (`) characters. `ROW FORMAT DELIMITED` means that Athena will use a default library called `LazySimpleSerDe` (p. 150) to do the actual work of parsing the data. The example also specifies that the fields are tab separated (`FIELDS TERMINATED BY '\t'`) and that each record in the file ends in a newline character (`LINES TERMINATED BY '\n'`). 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. 138) 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. 134).

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**

1. For **Database**, make sure that `mydatabase` is selected.
2. Choose the plus (+) sign in the Query Editor to create a tab with a new query. You can have up to ten query tabs open at once.
3. 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.

```sql
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+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+([^ \s]+)\s+[^%20]+[\s\S]*%20$"  
)  
LOCATION 's3://athena-examples-myregion/cloudfront/plaintext/';
```

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

5. Choose Run Query.

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. 472).

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 Query.

The results look like the following:
3. You can save the results of the query to a .csv file by choosing the download icon on the Results pane.

4. Choose the History tab above the Query Editor to view your previous queries.

5. Choose Download results to download the results of a previous query. Query history is retained for 45 days.
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. 19). 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. 18).

For more information, see Working with Query Results, Output Files, and Query History (p. 162).
Accessing Amazon Athena

You can access Amazon Athena using the AWS Management Console, the Amazon Athena API, or the AWS CLI.

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. 544).

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. 18)
- Using Athena Data Connector for External Hive Metastore (p. 36)
- Using Amazon Athena Federated Query (p. 68)
- IAM Policies for Accessing Data Catalogs (p. 80)
- Managing Data Sources (p. 84)
- Connecting to Amazon Athena with ODBC and JDBC Drivers (p. 86)

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 Data Catalogs (p. 341).
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. 35) For more information about the benefits of using AWS Glue with Athena, see Why should I upgrade to the AWS Glue Data Catalog? (p. 34)

**Topics**
- Using AWS Glue to Connect to Data Sources in Amazon S3 (p. 19)
- Registering an AWS Glue Data Catalog from Another Account (p. 22)
- Best Practices When Using Athena with AWS Glue (p. 23)
- Upgrading to the AWS Glue Data Catalog Step-by-Step (p. 31)
- FAQ: Upgrading to the AWS Glue Data Catalog (p. 34)

**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 set up an AWS Glue crawler to retrieve the information automatically, or you can manually add a table and enter the schema information.

**Setting up a Crawler**

You set up a crawler by starting in the Athena console and then using the AWS Glue console in an integrated way. When you create a crawler, you can choose data stores to crawl or point the crawler to existing catalog tables.

**Note**
The steps for setting up a crawler depend on the options available in the Athena console. If the Connect data source link in Option A is not available, use the procedure in Option B.

**Option A**

**Option A: To set up a crawler in AWS Glue using the Connect data source link**

2. Choose Connect data source. If the Connect data source link is not present, use Option B.
3. On the **Connect data source** page, choose **AWS Glue Data Catalog**.
4. Click **Next**.
5. On the **Connection details** page, choose **Set up crawler in AWS Glue to retrieve schema information automatically**.
6. Click **Connect to AWS Glue**.
7. On the **AWS Glue console** **Add crawler** page, follow the steps to create a crawler.

   For more information, see [Populating the AWS Glue Data Catalog](#).

### Option B

Use the following procedure to set up a AWS Glue crawler if the **Connect data source** link in **Option A** is not available in the Athena console.

**Option B: To set up a crawler in AWS Glue from the **AWS Glue Data Catalog** link**

2. Choose **AWS Glue Data Catalog**.
3. On the AWS Glue console **Tables** page, choose **Add tables using a crawler**.
4. On the **AWS Glue** console **Add crawler** page, follow the steps to create a crawler.

For more information, see [Populating the AWS Glue Data Catalog](#).

**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 Schema Table Manually

The following procedure shows you how to use the Athena console to add a table manually.

**To add a table and enter schema information manually**

2. Choose **Connect data source**.
3. On the **Connect data source** page, choose **AWS Glue Data Catalog**.
4. Click **Next**.
5. On the **Connection details** page, choose **Add a table and enter schema information manually**.
6. Click **Continue to add table**.
7. On the **Add table** page of the Athena console, for **Database**, choose an existing database or create a new one.
8. Enter or choose a table name.
9. For **Location of Input Data Set**, specify the path in Amazon S3 to the folder that contains the dataset that you want to process.
10. Click **Next**.
11. For **Data Format**, choose a data format (**Apache Web Logs, CSV, TSV, Text File with Custom Delimiters, JSON, Parquet**, or **ORC**).
   - For the **Apache Web Logs** option, you must also enter a regex expression in the **Regex** box.
   - For the **Text File with Custom Delimiters** option, specify a **Field terminator** (that is, a column delimiter). Optionally, you can specify a **Collection terminator** for array types or a **Map key terminator**.
12. For **Columns**, specify a column name and the column data type.

   - 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**.

13. Choose **Next**.

14. (Optional) For **Partitions**, click **Add a partition** to add column names and data types.

15. Choose **Create table**. The DDL for the table that you specified appears in the **Query Editor**. The following example shows the DDL generated for a two-column table in CSV format:

   ```sql
   CREATE EXTERNAL TABLE IF NOT EXISTS MyManualDB.MyManualTable (
   `cola` string,
   `colb` string
   )
   ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'
   WITH SERDEPROPERTIES (
   'serialization.format' = ',
   'field.delim' = ',
   )
   LOCATION 's3://bucket_name/
   TABLEPROPERTIES ('has_encrypted_data'=false);
   ```

16. Choose **Run query** to 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 Data Catalogs (p. 341).

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 Data Catalogs (p. 341) to ensure that you have permissions to query the data catalog in the other account.
3. Choose **Data sources**.
4. Choose **Connect data source**.
5. Choose **Next**.
6. Choose **AWS Glue Data Catalog in another account**.
7. Choose **Next**.
8. On the **Connection details: AWS AWS Glue Data Catalog in another account** page, enter the following information:
   - **Catalog 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. Choose **Register**. On the **Data sources** page, the data catalog that you entered is listed in the **Catalog name** column.

10. To view or edit information about the new data catalog, choose the catalog, and then choose **Edit**.

11. To delete the new data catalog, choose the catalog, and then choose **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. 23)**
- **Using AWS Glue Crawlers (p. 24)**
  - Scheduling a Crawler to Keep the AWS Glue Data Catalog and Amazon S3 in Sync (p. 24)
  - Using Multiple Data Sources with Crawlers (p. 24)
  - Syncing Partition Schema to Avoid "HIVE_PARTITION_SCHEMA_MISMATCH" (p. 26)
  - Updating Table Metadata (p. 27)
- **Working with CSV Files (p. 27)**
  - CSV Data Enclosed in Quotes (p. 27)
  - CSV Files with Headers (p. 29)
- **Working with Geospatial Data (p. 29)**
- **Using AWS Glue Jobs for ETL with Athena (p. 29)**
  - Creating Tables Using Athena for AWS Glue ETL Jobs (p. 30)
  - Using ETL Jobs to Optimize Query Performance (p. 31)
  - Converting SMALLINT and TINYINT Datatypes 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 252 characters.
- A table name cannot be longer than 255 characters.
- A column name cannot be longer than 128 characters.
- The only acceptable characters for database names, table names, and column names are lowercase letters, numbers, and the underscore character.

You can use the AWS Glue Catalog Manager to rename columns, but at this time table names and database names cannot be changed using the AWS Glue console. To correct database names, you need to
create a new database and copy tables 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 and register the schema for datasets in the AWS Glue Data Catalog. The crawlers go through your data, and inspect portions of it to determine the schema. In addition, the crawler can detect and register partitions. For more information, see Cataloging Data with a Crawler 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.

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 another 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. Choose **Crawlers**, select your crawler, and then choose **Action, Edit crawler**.

3. Under **Add information about your crawler**, choose additional settings as appropriate, and then choose **Next**.
4. Under **Add a data store**, change **Include path** to the table-level directory. For instance, given the example above, you would change it from s3://bucket01/folder1 to s3://bucket01/folder1/table1/. Choose **Next**.
5. For **Add another data store**, choose **Yes, Next**.

6. For **Include path**, enter your other table-level directory (for example, `s3://bucket01/folder1/table2/`) and choose **Next**.
   
   a. Repeat steps 3-5 for any additional table-level directories, and finish the crawler configuration.

The new values for **Include locations** appear under data stores as follows:

**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. 24).

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](https://docs.aws.amazon.com/glue/latest/dg/monitor-table-metadata.html).

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. 27).

**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 OpenCSVSerDe, see OpenCSVSerDe for Processing CSV (p. 140).

**To edit table properties in the AWS Glue console**

1. In the AWS Glue console navigation pane, choose Tables.
2. Choose the table that you want to edit, and then choose Edit table.
3. In the Edit table details dialog box, make the following changes:
   - For **Serde 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 (`\"`).
     - 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 Programmatically

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": "\\",
        "escapeChar": "\\"
    }
},
```

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 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:

```
 glueContext.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")
```

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. 24) and Working with CSV Files (p. 27). 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. 223).

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 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 csv, parquet, orc, avro, or json. 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 change the classification property using the console

1. Choose Edit Table.
2. For Classification, select the file type and choose Apply.

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. 34).

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. 23).

**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/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 IAM Policies for User Access (p. 325).

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "glue:CreateDatabase",
                "glue:DeleteDatabase",
                "glue:ImportCatalogToGlue"
            ],
            "Resource": [ "*" ]
        }
    ]
}
```
"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: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. After you upgrade, the Athena console has a link to open the AWS Glue Catalog Manager from within Athena.

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. 24)**.
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](https://docs.aws.amazon.com/glue/latest/dg/what-is.html).

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. 35)
- My users use a managed policy with Athena and Redshift Spectrum. What steps do I need to take to upgrade? (p. 36)
- What happens if I don’t upgrade? (p. 36)
- Why do I need to add AWS Glue policies to Athena users? (p. 36)
- What happens if I don’t allow AWS Glue policies for Athena users? (p. 36)
- Is there risk of data loss during the upgrade? (p. 36)
- Is my data also moved during this upgrade? (p. 36)

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/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. 37)
- Workflow (p. 37)
- Considerations and Limitations (p. 38)
- Connecting Athena to an Apache Hive Metastore (p. 39)
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. 68).
- **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. 177) 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.
• Hive views are not compatible with Athena views and are not supported.
• Kerberos authentication for Hive metastore is not supported.
• To use the JDBC driver with an external Hive metastore or federated queries (p. 68), include MetadataRetrievalMethod=ProxyAPI in your JDBC connection string. For information about the JDBC driver, see Using Athena with the JDBC Driver (p. 87).

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. 348) and Allow Lambda Function Access to External Hive Metastores (p. 350).

• 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. 39) 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. 326). For more information, see Identity and Access Management in Athena (p. 324).

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. 37) 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. 48).

To connect Athena to a Hive metastore

2. Choose **Connect data source**.

3. On the **Connect data source** page, for **Choose a metadata catalog**, choose **Apache Hive metastore**.

4. Choose **Next**.
5. On the **Connection details** page, for **Lambda function**, choose **Configure new AWS Lambda function**.

The `AthenaHiveMetastoreFunction` page opens in the AWS Lambda console.
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.
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.

8. Return to the Connection details page of the Data Sources tab in the Athena console.

9. Choose the Refresh icon next to Choose Lambda function. Refreshing the list of available functions causes your newly created function to appear in the list.
10. Now that your Lambda function is available, choose it.

A new **Lambda function ARN** entry shows the ARN of your Lambda function.
11. For **Catalog name**, enter a unique name that you will use in your SQL queries to reference the data source. The name can be up to 127 characters long and must be unique within your account. It cannot be changed after creation. Valid characters are a-z, A-z, 0-9, _(underscore), @(ampersand), and -(hyphen). The names awsdatacatalog, hive, jmx, and system are reserved by Athena and cannot be used for custom catalog names.

12. (Optional) For **Description**, enter text that describes your data catalog.

13. Choose **Connect**. This connects Athena to your Hive metastore catalog.

The **Data sources** page shows a list of your connected catalogs, including the catalog that you just connected. All registered catalogs are visible to all users in the same Amazon Web Services account.
14. You can now use the **Catalog name** that you specified to reference the Hive metastore in your SQL queries. In your SQL queries, use the following example syntax, replacing `hms-catalog-1` with the catalog name that you specified earlier.

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

15. To view, edit, or delete the data sources that you create, see [Managing Data Sources](#) (p. 84).

### Using the AWS Serverless Application Repository to Deploy a Hive Data Source Connector

You can also use the [AWS Serverless Application Repository](https://aws.amazon.com/serverless) to deploy an Athena data source connector for Hive. Choose the connector that you want to use, provide the parameters that the connector requires, and then deploy the connector to your account.

**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, type the name of one of the following connectors. 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.
• **AthenaHiveMetastoreFunction** – Uber Lambda function .jar file.
• **AthenaHiveMetastoreFunctionWithLayer** – Lambda layer and thin Lambda function .jar file.

5. Choose the name of the connector.

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 more information, see steps 8-15 of *Connecting Athena to an Apache Hive Metastore (p. 39)*.
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. 48) – Clone the Athena reference implementation and build the JAR file that contains the Lambda function code.
2. **AWS Lambda console** (p. 49) – 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. 54) – In the Amazon Athena console, create a data catalog 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. 39).

**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. 350).

**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. 67) topic in this documentation.

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

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

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

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

   ```
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/](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`).
10. Expand **Advanced settings**.

11. For **VPC**, choose the VPC that your function will have access to.

12. For **Subnets**, choose the VPC subnets for Lambda to use.

13. For **Security groups**, choose the VPC security groups for Lambda to use.

14. Choose **Create function**. The AWS Lambda console and 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 console, navigate to the page for your function if necessary.
2. For Function code, choose Actions, and then choose Upload a .zip or .jar file.
3. Upload the `hms-lambda-func-1.0-SNAPSHOT-withdep.jar` file that you generated previously.

4. In the Environment variables section of the configuration page for your function, choose Edit.

5. On the Edit environment variables page, 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.
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.

6. Choose **Save**.

**Connect Athena to Your Hive Metastore**

Now you can use the Athena console to prepare Athena to use your Hive metastore. In this step, you create a data catalog name to use in your Athena queries that refers to your external Hive metastore.

**To connect Athena to your Hive metastore**

2. Do one of the following:
   - In the Query Editor navigation pane, choose **Connect data source**.
Choose the Data sources tab, and then choose Connect data source.

3. On the Connect data source page, for Choose a metadata catalog, choose Apache Hive metastore.
4. Choose **Next**.

5. On the **Connection details** page, for **Lambda function**, use the **Choose Lambda function** option to choose the Lambda function that you created.

A new **Lambda function ARN** entry shows the ARN of your Lambda function.
6. For **Catalog name**, enter a unique name that you will use in your SQL queries to reference your Hive data source.

   **Note**  
   The names awsdatalake, hive, jmx, and system are reserved by Athena and cannot be used for custom catalog names.

7. Choose **Connect**. This connects Athena to your Hive metastore catalog.

8. You can now use the **Catalog name** that you specified to reference the Hive metastore in your SQL queries. In your SQL queries, use the following example syntax, replacing `ehms-catalog` with the catalog name that you specified earlier.

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

9. To view, edit, or delete the data sources that you create, see Managing Data Sources (p. 84).
Using a Default Catalog 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 catalog name before the database using the syntax

   `[[catalog_name].database_name].table_name`

   as in the following example.

   ```sql
   select * from "hms-catalog-1".hms_tpch.customer limit 10;
   ```

2. When the catalog that you want to use is already selected as your data source, you can omit the catalog name from the query, as in the following example.

   ```sql
   select * from hms_tpch.customer limit 10;
   ```
3. For multiple catalogs, you can omit only the default catalog name. Specify the full name for any non-default catalogs. For example, the FROM statement in the following query omits the catalog name for AWS Glue catalog, but it fully qualifies the first two catalog names.

... 
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.

```
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, when you select the catalog and the database in the Data source panel, you can omit the catalog prefix from the query.

In the following example, the data source and database are selected in the query editor. The show create table customer statement succeeds when the hms-catalog-1 prefix and the hms_tpch database name are omitted from the query.
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=key,Value=value` argument pairs.

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

```bash
$ 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.

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

The following sample result is in JSON format.

```json
{
   "DataCatalog": {
      "Name": "hms-catalog-1",
      "Description": "Hive Catalog 1",
      "Type": "HIVE",
      "Parameters": {
         "sdk-version": "1.0"
      }
   }
}
```

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": "AwsDataCatalog",
```
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.

```
$ 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.

```
$ 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.

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

The following sample result is in JSON format.

```
{
   "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.
$ aws athena list-databases --catalog-name AwsDataCatalog --region us-west-2

The following sample result is in JSON format.

```
{
  "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",
```
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"
      }
    }
  ]
}
```
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
```
DML Statements

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

```bash
$ 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.

```bash
$ 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 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. 72).
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. For information about Athena engine versions, see Athena Engine Versioning (p. 459). For a list of AWS Regions that support Athena engine version 2, see Athena engine version 2 (p. 463).

- **Views** – You cannot use views with federated data sources.

- **Pricing** – For pricing information, see Amazon Athena pricing.

  **JDBC driver** – To use the JDBC driver with federated queries or an external Hive metastore (p. 36), include `MetadataRetrievalMethod=ProxyAPI` in your JDBC connection string. For information about the JDBC driver, see Using Athena with the JDBC Driver (p. 87).

- **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. 354).

- **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.

**Analyze Results of Federated Query in Amazon Athena 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.

**Game Analytics Pipeline**
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 the first part, you give the Lambda function a name that you can later choose in the Athena console. In the second part, you 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. 459).

**To deploy a data source connector**

2. Do one of the following:
   - In the Query Editor navigation pane, choose **Connect data source**.
   - Choose the **Data sources** tab, and then choose **Connect data source**.
3. On the **Connect data source** page, choose **Query a data source**.
4. For **Choose a data source**, choose the data source that you want to query with Athena, such as **Amazon CloudWatch Logs**.
5. Choose **Next**.
6. For **Lambda function**, choose **Configure new AWS Lambda function**. The function page for the connector that you chose opens in the Lambda console. The page includes detailed information about the connector.
7. Under **Application settings**, read the description for each application setting carefully, and then enter values that correspond to your requirements.
   
   **Note**
   The application settings that you see vary depending on the connector for your data source. The Amazon Redshift, MySQL, and PostgreSQL options use the Amazon Athena connector for JDBC-compliant data sources. For more information, see [Amazon Athena Lambda JDBC Connector on GitHub](https://github.com/aws-amazon-athena-lambda-jdbc). The minimum required settings include:
   
   - **AthenaCatalogName** – A name 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.

8. Select **I acknowledge that this app creates custom IAM roles**. For more information, choose the **Info** link.
9. Choose **Deploy**. The **Resources** section of the Lambda console shows the deployment status of the connector and informs you when the deployment is complete.

---

**Part 2: Connecting to a Data Source**

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

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

2. Choose **Connect data source**.
   
   Do one of the following:
   
   - In the Query Editor navigation pane, choose **Connect data source**.
   - Choose the **Data sources** tab, and then choose **Connect data source**.
3. Choose **Query a data source**.
4. Choose the data source for the connector that you just deployed, such as Amazon CloudWatch Logs. If you used the Athena Query Federation SDK to create your own connector and have deployed it to your account, choose All other data sources.

5. Choose Next.

6. For Choose Lambda function, choose the function that you named. The Lambda function's ARN is displayed.

7. For Catalog name, enter a unique name to use for the data source in your SQL queries, such as cloudwatchlogs. The name can be up to 127 characters and must be unique within your account. It cannot be changed after creation. Valid characters are a-z, A-z, 0-9, _(underscore), @(ampersand) and -(hyphen). The names awsdatalcataloq, hive, jmx, and system are reserved by Athena and cannot be used for custom catalog names.

8. Choose Connect. The Data sources page now shows your connector in the list of catalog names. You can now use the connector in your queries.

For information about writing queries with data connectors, see Writing Federated Queries (p. 76).

Using the AWS Serverless Application Repository to Deploy a Data Source Connector

You can also use the AWS Serverless Application Repository to deploy an Athena data source connector. You find the connector that you want to use, provide the parameters that the connector requires, and then deploy 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. 459).

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, or search for applications published with the author name Amazon Athena Federation. This author name is reserved for applications that the Amazon Athena team has written, tested, and validated. For a list of prebuilt Athena data connectors, see Using Athena Data Source Connectors (p. 72).

5. Choose the name of the connector. This 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, SpillBucket, specify an Amazon S3 bucket to receive data from large response payloads. For information about the remaining configurable options, see the corresponding Available Connectors topic on GitHub.

7. At the bottom right of the Application details page, choose Deploy.

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.

Notes
Before you start, note the following points:

- To use the Amazon Athena Federated Query feature, set your workgroup to Athena engine version 2. For steps, see Changing Athena Engine Versions (p. 459).
- 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.

For more information about data source connectors, see the following topics:

- For information about deploying an Athena data source connector, see Deploying a Connector and Connecting to a Data Source (p. 70).
- For information about writing queries that use Athena data source connectors, see Writing Federated Queries (p. 76).
- For in-depth information about the Athena data source connectors, see Available Connectors on GitHub.

Topics

- Amazon Athena CloudWatch Connector  (p. 73)
- Amazon Athena CloudWatch Metrics Connector (p. 73)
- Athena AWS CMDB Connector (p. 74)
- Amazon Athena DocumentDB Connector (p. 74)
- Amazon Athena DynamoDB Connector (p. 74)
- Amazon Athena Elasticsearch Connector (p. 74)
- Amazon Athena HBase Connector (p. 74)
- Amazon Athena Connector for JDBC-Compliant Data Sources (PostgreSQL, MySQL, and Amazon Redshift) (p. 74)
- Amazon Athena Neptune Connector (p. 75)
- Amazon Athena Redis Connector (p. 75)
- Amazon Athena Timestream Connector (p. 75)
- Amazon Athena TPC Benchmark DS (TPC-DS) Connector (p. 75)
- Amazon Athena Vertica Connector (p. 75)

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.

For more information about configuration options, throttling control, table mapping schema, permissions, deployment, performance considerations, and licensing, see Amazon Athena CloudWatch Connector on GitHub.

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.
For information about configuration options, table mapping, permissions, deployment, performance considerations, and licensing, see Amazon Athena Cloudwatch Metrics Connector on GitHub.

**Athena AWS CMDB Connector**

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

For information about supported services, parameters, permissions, deployment, performance, and licensing, see Amazon Athena AWS CMDB Connector on GitHub.

**Amazon Athena DocumentDB Connector**

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

For information about how the connector generates schemas, configuration options, permissions, deployment, and performance considerations, see Amazon Athena DocumentDB Connector on GitHub.

**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.

For information about configuration options, permissions, deployment, and performance considerations, see Amazon Athena DynamoDB Connector on GitHub.

**Amazon Athena Elasticsearch Connector**

The Amazon Athena Elasticsearch connector enables Amazon Athena to communicate with your Elasticsearch instances so that you can use SQL to query your Elasticsearch data. This connector works with the Amazon Elasticsearch Service or any Elasticsearch-compatible endpoint that is configured with Elasticsearch version 7.0 or higher.

For information about configuration options, databases and tables, data types, deployment, and performance considerations, see Amazon Athena Elasticsearch Connector on GitHub.

**Amazon Athena HBase Connector**

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

For information about configuration options, data types, permissions, deployment, performance, and licensing, see Amazon Athena HBase Connector on GitHub.

**Amazon Athena Connector for JDBC-Compliant Data Sources (PostgreSQL, MySQL, and Amazon Redshift)**

The Amazon Athena Lambda JDBC connector enables Amazon Athena to access your JDBC-compliant database. Currently supported databases include MySQL, PostgreSQL, and Amazon Redshift.

For information about supported databases, configuration parameters, supported data types, JDBC driver versions, limitations, and other information, see Amazon Athena Lambda JDBC Connector on GitHub.
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.

For information about configuration options, permissions, deployment, performance, and limitations, see Amazon Athena Neptune Connector on GitHub.

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.

For information about configuration options, setting up databases and tables, data types, permissions, deployment, performance, and licensing, see Amazon Athena Redis Connector on GitHub.

Amazon Athena Timestream Connector

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. For more information, see the Amazon Timestream Developer Guide.

The Amazon Athena Timestream connector enables Amazon Athena to communicate with Amazon Timestream timeseries data. You can optionally use AWS Glue Data Catalog as a source of supplemental metadata.

For information about configuration options, setting up databases and tables, data types, permissions, deployment, performance, and licensing, see Amazon Athena Timestream Connector on GitHub.

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. The Athena TPC-DS connector generates a TPC-DS compliant database at one of four scale factors.

For information about configuration options, databases and tables, permissions, deployment, performance, and licensing, see Amazon Athena TPC-DS Connector on GitHub.

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.

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 on GitHub.

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.

### 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`.

```
-- 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),
payments AS (SELECT "summary:order_id",
```
"summary:status",
"summary:cc_id",
"details:network"
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,
publicipaddress AS ec2_order_processor,
NAME AS ec2_state,
log_level,
order_processor_log
FROM active_orders
LEFT JOIN logs
  ON logs.orderid = active_orders._key_
LEFT JOIN order_processors
  ON logs.log_stream = order_processors.instanceid
LEFT JOIN customer
  ON customer.id = customer_id
LEFT JOIN addresses
  ON addresses.id = address_id
LEFT JOIN shipments
  ON shipments.order_id = active_orders._key_
LEFT JOIN payments
  ON payments."summary:order_id" = active_orders._key_

The following image shows sample results of the query.

![Sample query results](image)

**Writing a Data Source Connector Using the Athena Query Federation SDK**

To write your own data source connectors (p. 68), 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.
**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. 459).

For information about how to download the SDK and detailed instructions for writing your own connector, see Example Athena Connector on GitHub.

## 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. 81).

**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. 81).
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. 81).
For information on 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. 80).

- Example Policy for Full Access to All Data Catalogs (p. 81)
- Example Policy for Full Access to a Specified Data Catalog (p. 81)
- Example Policy for Querying a Specified Data Catalog (p. 82)
- Example Policy for Management Operations on a Specified Data Catalog (p. 83)
- Example Policy for Listing Data Catalogs (p. 83)
- Example Policy for Metadata Operations on Data Catalogs (p. 84)

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.

```
{
   "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.
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:

```
{
  "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 view, edit, or delete the data sources that you create, including Athena data source connector, AWS Glue Data Catalog, and Hive metastore catalog types.

To view a data source

- Do one of the following:
  - Choose the catalog name of the data source.
  - Select the button next to the catalog name, and then choose View details.

The details page includes options to Edit or Delete the data source.

To edit a data source

1. Do one of the following:
   - Select the button next to the catalog name, and then choose Edit.
   - Choose the catalog name of the data source, and then choose Edit.
2. On the Edit page for the metastore, you can choose a different Lambda function for the data source or change the description of the existing function. When you edit an AWS Glue catalog, the AWS Glue console opens the corresponding catalog for editing.
3. Choose **Save**.

**To delete a data source**

1. Select the button next to the data source or the name of the data source, and then choose **Delete**. You are warned that when you delete a metastore data source, its corresponding Data Catalog, tables, and views are removed from the query editor. Saved queries that used the metastore no longer run in Athena.
2. 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. 87)
- Connecting to Amazon Athena with ODBC (p. 88)
- Using the Amazon Athena Power BI Connector (p. 90)

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?
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, download, install, and configure the Athena JDBC driver, using the following links on Amazon S3. For permissions information, see Access through JDBC and ODBC Connections (p. 330).

Links for Downloading the JDBC Driver

The JDBC driver version 2.0.23 complies with the JDBC API 4.1 and 4.2 data standards. Before downloading the driver, check which version of Java Runtime Environment (JRE) you use. The JRE version depends on the version of the JDBC API you are using with the driver. If you are not sure, download the latest version of the driver.

Download the driver that matches your version of the JDK and the JDBC data standards:

- The AthenaJDBC41.jar is compatible with JDBC 4.1 and requires JDK 7.0 or later.
- The AthenaJDBC42.jar is compatible with JDBC 4.2 and requires JDK 8.0 or later.

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.

Important

To use the JDBC driver for multiple data catalogs with Athena (for example, when using an external Hive metastore (p. 36) or federated queries (p. 68)), include MetadataRetrievalMethod=ProxyAPI in your JDBC connection string.

Migration from Previous Version of the JDBC Driver

The current JDBC driver version 2.0.23 is a drop-in replacement of the previous version of the JDBC driver version 2.0.9, and is backwards compatible with the JDBC driver version 2.0.9, with the following step that you must perform to ensure the driver runs.

Important

To use JDBC driver version 2.0.5 or later, attach a permissions policy to IAM principals using the JDBC driver that allows the athena:GetQueryResultsStream policy action. This policy
Connecting to Amazon Athena with ODBC

Download the Amazon Athena ODBC driver License Agreement, ODBC drivers, and ODBC documentation using the following links. For permissions information, see Access through JDBC and ODBC Connections (p. 330).

**Amazon Athena ODBC Driver License Agreement**

**ODBC Driver Download Links**

**Windows**

<table>
<thead>
<tr>
<th>Driver Version</th>
<th>Download Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC 1.1.12 for Windows 32-bit</td>
<td>Windows 32 bit ODBC Driver 1.1.12</td>
</tr>
<tr>
<td>ODBC 1.1.12 for Windows 64-bit</td>
<td>Windows 64 bit ODBC Driver 1.1.12</td>
</tr>
</tbody>
</table>

**Linux**

<table>
<thead>
<tr>
<th>Driver Version</th>
<th>Download Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC 1.1.12 for Linux 32-bit</td>
<td>Linux 32 bit ODBC Driver 1.1.12</td>
</tr>
<tr>
<td>ODBC 1.1.12 for Linux 64-bit</td>
<td>Linux 64 bit ODBC Driver 1.1.12</td>
</tr>
</tbody>
</table>

**OSX**

<table>
<thead>
<tr>
<th>Driver Version</th>
<th>Download Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC 1.1.12 for OSX</td>
<td>OSX ODBC Driver 1.1.12</td>
</tr>
</tbody>
</table>
Documentation

<table>
<thead>
<tr>
<th>Driver Version</th>
<th>Download Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documentation for ODBC 1.1.12</td>
<td>ODBC Driver Installation and Configuration Guide version 1.1.12</td>
</tr>
</tbody>
</table>

**Migration from the Previous Version of the ODBC Driver**

The current ODBC driver version 1.1.12 is a drop-in replacement of the previous version of the ODBC driver version 1.0.5. It is also backward compatible with the ODBC driver version 1.0.3, if you use the following required steps to make sure that the driver runs.

**Important**

To use the ODBC driver versions 1.0.3 and greater, follow these requirements:

- 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.
- Add the `athena:GetQueryResultsStream` policy action to the list of policies for Athena. This policy action is not exposed directly with the API operation, and is used only with the ODBC and JDBC drivers, as part of streaming results support. For an example policy, see AWSQuicksightAthenaAccess Managed Policy (p. 328).

**Previous Versions of the ODBC Driver**

<table>
<thead>
<tr>
<th>Driver Version 1.0.5</th>
<th>Download Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC 1.0.5 for Windows 32-bit</td>
<td>Windows 32 bit ODBC Driver 1.0.5</td>
</tr>
<tr>
<td>ODBC 1.0.5 for Windows 64-bit</td>
<td>Windows 64 bit ODBC Driver 1.0.5</td>
</tr>
<tr>
<td>ODBC 1.0.5 for Linux 32-bit</td>
<td>Linux 32 bit ODBC Driver 1.0.5</td>
</tr>
<tr>
<td>ODBC 1.0.5 for Linux 64-bit</td>
<td>Linux 64 bit ODBC Driver 1.0.5</td>
</tr>
<tr>
<td>ODBC 1.0.5 for OSX</td>
<td>OSX ODBC Driver 1.0.5</td>
</tr>
<tr>
<td>Documentation for ODBC 1.0.5</td>
<td>ODBC Driver Installation and Configuration Guide version 1.0.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Driver Version 1.0.4</th>
<th>Download Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC 1.0.4 for Windows 32-bit</td>
<td>Windows 32 bit ODBC Driver 1.0.4</td>
</tr>
<tr>
<td>ODBC 1.0.4 for Windows 64-bit</td>
<td>Windows 64 bit ODBC Driver 1.0.4</td>
</tr>
<tr>
<td>ODBC 1.0.4 for Linux 32-bit</td>
<td>Linux 32 bit ODBC Driver 1.0.4</td>
</tr>
<tr>
<td>ODBC 1.0.4 for Linux 64-bit</td>
<td>Linux 64 bit ODBC Driver 1.0.4</td>
</tr>
<tr>
<td>ODBC 1.0.4 for OSX</td>
<td>OSX ODBC Driver 1.0.4</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.

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. 330)
- Amazon Athena ODBC driver (p. 88)
- 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**.
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. 89).

- 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. 97)
- Creating Tables in Athena (p. 99)
- Names for Tables, Databases, and Columns (p. 106)
- Reserved Keywords (p. 107)
- Table Location in Amazon S3 (p. 109)
- Columnar Storage Formats (p. 110)
- Converting to Columnar Formats (p. 111)
- Partitioning Data in Athena (p. 114)
- Partition Projection with Amazon Athena (p. 119)

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 Query Editor tab, enter the Hive data definition language (DDL) command `CREATE DATABASE myDatabase`. Replace `myDatabase` with the name of the database that you want to create.
3. Choose Run Query or press Ctrl+ENTER.
4. To make your database the current database, select it from the Database menu.

Note
Currently, the database menu can show and filter a maximum of 1000 databases regardless of the type of data catalog used. This is a limitation of the Athena console, not an account limit. The number of databases that can be displayed and filtered is not user configurable.
Creating Tables in Athena

You can run DDL statements in the Athena console, using a JDBC or an ODBC driver, or using the Athena Add table wizard (p. 102).

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. 134).

**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. 331).
- 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 Creating a Workgroup (p. 430).
- Athena does not support querying the data in the S3 Glacier or S3 Glacier Deep Archive storage classes. Objects in the S3 Glacier storage class are ignored. Objects in the S3 Glacier Deep Archive storage class that are queried result in the error message The operation is not valid for the object’s storage class. 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.

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 Developer 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.
Functions Supported

The functions supported in Athena queries are those found within Presto. For more information, see the documentation for Presto versions 0.172 and 0.217, which correspond to Athena engine versions 1 and 2.

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. 503).

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.

All Tables Are EXTERNAL

If you use CREATE TABLE without the EXTERNAL keyword, Athena issues an error; only tables with the EXTERNAL keyword can be created. We recommend that you always use the EXTERNAL keyword. 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 wizard, or by running a DDL statement in the Athena Query Editor.

To create a table using the AWS Glue Data Catalog

2. In the Query Editor, under Database, choose Create table, and then choose from AWS Glue crawler.
3. In the Go to AWS Glue to set up a crawler dialog box, choose Continue.

You will go to the AWS Glue console to set up a crawler. The crawler connects to your data store and automatically determines its structure to create the metadata for your table.

Do not show this message again

4. Follow the steps in the AWS Glue console to add a crawler.

For more information, see Using AWS Glue Crawlers (p. 24).

To create a table using the Athena add table wizard

2. Under the database display in the Query Editor, choose **Create table**, and then choose **from S3 bucket data**.

3. In the **Add table** wizard, follow the steps to create your table.
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, and then choose Run Query, or press Ctrl+ENTER.

```sql
CREATE EXTERNAL TABLE IF NOT EXISTS cloudfront_logs (  `Date` Date,  `Time` STRING,  `Location` STRING,  `Bytes` INT,  `RequestIP` STRING,  `Method` STRING,  `Host` STRING,  `Uri` STRING,
```
Showing Table Information

After the table is created, you can run queries against your data.

After you have created a table in Athena, you can choose the vertical three dots next to the table name in the Athena console to show information about the table and manage it.

- **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.
- **Show properties** – Displays table properties and, if the table has partitions, partition information on a second tab.
  - **Table properties** – Shows the table name, database name, time created, and whether the table has encrypted data.
  - **Partitions** – Lists the names of the table's partitions and their corresponding data types. This tab displays only if the table has partitions.
- **Delete table** – Displays a confirmation dialog box asking if you want to delete the table. If you agree, runs the `DROP TABLE (p. 519) table_name` statement in the Athena Query Editor.
- **Load partitions** – Runs the `MSCK REPAIR TABLE (p. 520) table_name` statement in the Athena Query Editor. This option is available only if the table has partitions.
- **Generate Create Table DDL** – Generates a DDL statement that you can use to re-create the table by running the `SHOW CREATE TABLE (p. 522) table_name` statement in the Athena Query Editor.

**Note**

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
Names for Tables, Databases, and Columns

Use these tips for naming items in Athena.

Table names and table column names in Athena must be lowercase

If you are interacting with Apache Spark, then your table names and table column names must be lowercase. Athena is case-insensitive and turns table names and column names to lower case, but Spark requires lowercase table and column names.

Queries with mixedCase column names, such as `profileURI`, or upper case column names do not work.

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:

```sql
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:

```sql
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. 172), enclose them in double quotes ("').

For more information, see Reserved Keywords (p. 107).

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. 172), enclose them in double quotes ("').

- List of Reserved Keywords in DDL Statements (p. 107)
- List of Reserved Keywords in SQL SELECT Statements (p. 107)
- Examples of Queries with Reserved Keywords (p. 108)

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 (`).

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.

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, UN ESCAPE, UNION, UNNEST, USING, VALUES, WHEN, WHERE, WITH

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:

```
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 (`):

```
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:

```
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:

```
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 Developer Guide.
- For information about using folders in Amazon S3, see Using Folders in the Amazon Simple Storage Service Console 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. 110).

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 file names, 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`
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. 24).
- You can also create partitions in a table directly in Athena. For more information, see Partitioning Data (p. 114).

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.
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.

Note
Use the CREATE TABLE AS (CTAS) (p. 184) queries to perform the conversion to columnar formats, such as Parquet and ORC, in one step.

You can do this to existing Amazon S3 data sources by creating a cluster in Amazon EMR and converting it using Hive. The following example using the AWS CLI shows you how to do this with a script and data stored in Amazon S3.

Overview

The process for converting to columnar formats using an EMR cluster is as follows:

1. Create an EMR cluster with Hive installed.
2. In the step section of the cluster create statement, specify a script stored in Amazon S3, which points to your input data and creates output data in the columnar format in an Amazon S3 location. In this example, the cluster auto-terminates.

Note
The script is based on Amazon EMR version 4.7 and needs to be updated to the current version. For information about versions, see Amazon EMR Release Guide.

The full script is located on Amazon S3 at:

s3://athena-examples-myregion/conversion/write-parquet-to-s3.q

Here's an example script beginning with the CREATE TABLE snippet:

```
ADD JAR /usr/lib/hive-hcatalog/share/hcatalog/hive-hcatalog-core-1.0.0-amzn-5.jar;
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'
with serdeproperties ( 'paths'='requestBeginTime, adId, impressionId, referrer, userAgent, userCookie, ip' )
```
LOCATION 's3://MyRegion.elasticmapreduce/samples/hive-ads/tables/impressions/';

**Note**
Replace *MyRegion* in the LOCATION clause with the region where you are running queries. For example, if your console is in us-west-1, s3://us-west-1.elasticmapreduce/samples/hive-ads/tables/.

This creates the table in Hive on the cluster which uses samples located in the Amazon EMR samples bucket.

3. On Amazon EMR release 4.7.0, include the `ADD JAR` line to find the appropriate JsonSerDe. The prettified sample data looks like the following:

```json
{
  "number": "977680",
  "referrer": "fastcompany.com",
  "processId": "1823",
  "adId": "TBktxshOXAHwo261jAHubijAoN1AgA",
  "browserCookie": "mvlrdwrmef",
  "userCookie": "emFlrGm5fA2xLFT5npwbPuG7kf6X",
  "requestEndTime": "1239714001000",
  "impressionId": "1I5G20RmOuG2rt7FPFgsaWk9Xpkfb",
  "userAgent": "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0; SLCC1; .NET CLR 2.0.50727; Media Center PC 5.0; .NET CLR 3.0.04506; InfoPath",
  "timer": {
    "modelLookup": "0.3292",
    "requestTime": "0.6398"
  },
  "threadId": "99",
  "ip": "67.189.155.225",
  "modelId": "bxxiuuxduad",
  "hostname": "ec2-0-51-75-39.amazon.com",
  "sessionId": "J9NOccA3DMFiliCuS01t9Q6Bjs6aS",
  "requestBeginTime": "1239714000000"
}
```

4. In Hive, load the data from the partitions, so the script runs the following:

```
MSCK REPAIR TABLE impressions;
```

The script then creates a table that stores your data in a Parquet-formatted file on Amazon S3:

```sql
CREATE EXTERNAL TABLE parquet_hive (
  requestBeginTime string,
  adId string,
  impressionId string,
  referrer string,
  userAgent string,
  userCookie string,
  ip string
) STORED AS PARQUET
LOCATION 's3://myBucket/myParquet/';
```

The data are inserted from the *impressions* table into *parquet_hive*:

```sql
INSERT OVERWRITE TABLE parquet_hive
SELECT
  requestBeginTime,
  adId,
  impressionId,
  referrer,
```
The script stores the above `impressions` table columns from the date, 2009-04-14-04-05, into s3://myBucket/myParquet/ in a Parquet-formatted file.

5. After your EMR cluster is terminated, create your table in Athena, which uses the data in the format produced by the cluster.

**Before you begin**

- You need to create EMR clusters. For more information about Amazon EMR, see the Amazon EMR Management Guide.
- Follow the instructions found in Setting Up (p. 6).

**Example: Converting data to Parquet using an EMR cluster**

1. Use the AWS CLI to create a cluster. If you need to install the AWS CLI, see Installing the AWS Command Line Interface in the AWS Command Line Interface User Guide.

2. You need roles to use Amazon EMR, so if you haven't used Amazon EMR before, create the default roles using the following command:

```bash
aws emr create-default-roles
```

3. Create an Amazon EMR cluster using the emr-4.7.0 release to convert the data using the following AWS CLI `emr create-cluster` command:

```bash
export REGION=us-west-1
export SAMPLEURI=s3://${REGION}.elasticmapreduce/samples/hive-ads/tables/impressions/
export S3BUCKET=myBucketName

aws emr create-cluster
   --applications Name=Hadoop Name=Hive Name=HCatalog \
   --ec2-attributes KeyName=myKey,InstanceProfile=EMR_EC2_DefaultRole,SubnetId=subnet-
   mySubnetId \
   --service-role EMR_DefaultRole \
   --release-label emr-4.7.0 \
   --instance-type m4.large \
   --instance-count 1 \
   --steps Type=HIVE,Name="Convert to Parquet", \
   ActionOnFailure=TERMINATE_CLUSTER, \
   Args=[-f, \
   "s3://athena-examples/conversion/write-parquet-to-s3.q",-hiveconf, \
   INPUT="${SAMPLEURI}",-hiveconf, \
   OUTPUT="s3://${S3BUCKET}/myParquet",-hiveconf, \
   REGION=${REGION} 
   } \
   --region #${REGION} \
   --auto-terminate
```

For more information, see Create and Use IAM Roles for Amazon EMR in the Amazon EMR Management Guide.

A successful request gives you a cluster ID.
4. Monitor the progress of your cluster using the AWS Management Console, or using the cluster ID with the `list-steps` subcommand in the AWS CLI:

```
aws emr list-steps --cluster-id myClusterID
```

Look for the script step status. If it is COMPLETED, then the conversion is done and you are ready to query the data.

5. Create the same table that you created on the EMR cluster.

You can use the same statement as above. Log into Athena and enter the statement in the Query Editor window:

```sql
CREATE EXTERNAL TABLE parquet_hive (  
    requestBeginTime string,
    adId string,
    impressionId string,
    referrer string,
    userAgent string,
    userCookie string,
    ip string
  ) STORED AS PARQUET
LOCATION 's3://myBucket/myParquet/';
```

Choose Run Query.

6. Run the following query to show that you can query this data:

```
SELECT * FROM parquet_hive LIMIT 10;
```

Alternatively, you can select the view (eye) icon next to the table's name in Catalog:

The results should show output similar to this:

---

**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. Athena leverages Apache Hive for partitioning data. 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 loaded one time per day, may partition by a data source identifier and date.

**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. 110).

- 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://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.

- Make sure that the Amazon S3 path is in lower case instead of camel case (for example, `userid` instead of `userId`). 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. 520).

- 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.

- If you are not using AWS Glue Data Catalog, the default maximum number of partitions per table is 20,000. You can request a quota increase.

**Creating and Loading a Table with Partitioned Data**

To create a table that uses partitions, you must define it during the `CREATE TABLE` (p. 511) statement. Use `PARTITIONED BY` to define the keys by which to partition data, as in the following example. `LOCATION` specifies the root location of the partitioned data.

```sql
CREATE EXTERNAL TABLE users (
  first string,
  last string,
  username string
) PARTITIONED BY (id string)
STORED AS parquet
LOCATION 's3://bucket/folder/
```

After you create the table, you load the data in the partitions for querying. For Hive-compatible data, you run `MSCK REPAIR TABLE` (p. 520). For non-Hive compatible data, you use `ALTER TABLE ADD PARTITION` (p. 505) to add the partitions manually.
Preparing Partitioned and Nonpartitioned Data for Querying

The following sections discuss two scenarios:

1. Data is already partitioned, stored on Amazon S3, and you need to access the data on Athena.
2. Data is not partitioned.

Scenario 1: Data already partitioned and stored on S3 in Hive format

Storing Partitioned Data

Partitions are stored in separate folders in Amazon S3. For example, here is the partial listing for sample ad impressions:

```
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.

Creating a Table

To make a table out of 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. 134).

After you run the preceding statement in Athena, choose New Query and run the following command:

```sql
MSCK REPAIR TABLE impressions
```

Athena loads the data in the partitions.

**Query the Data**

Now, query the data from the impressions table using the partition column. Here’s an example:

```sql
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 you data similar to the following:

```
2009-04-12-13-20    ap3HcVKAMfXtgIPu6WpuUfAfL0DQEc
2009-04-12-13-20    17uichtodoS9kkeQ1k0XThk151uRwV
2009-04-12-13-20    J0uf18CtRwvJ68v5CghqE5h0nk0tp
2009-04-12-13-20    N2XP0J0DvBcXJ0pb4Xvq35A4QxxH
2009-04-12-13-20    fPAIi1BrSggqro9Krdlwbex605Roaxr
2009-04-12-13-20    V4og4RW6G3GjH8WvgIiSc0qig5D1G
2009-04-12-13-20    hP8PtBwK45SmW4WxPV01kvV4v11b
2009-04-12-13-20    v0SfhexgeheD90gp31UCr6P1nFpx6ij
2009-04-12-13-20    ID9oDvQi1iOQ4XkkHMcOk87kKDKFj
2009-04-12-13-20    b31tJ1A25CK8eDHQrHNbcknfsNdUk
```

**Scenario 2: Data is not partitioned in Hive format**

A layout like the following does not, however, work for automatically adding partition data with MSCK REPAIR TABLE:

```bash
aws s3 ls s3://athena-examples-myregion/elb/plaintext/ --recursive
```

<table>
<thead>
<tr>
<th>Date</th>
<th>Size</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-11-23</td>
<td>11789573</td>
<td>elb/plaintext/2015/01/01/part-r-00000-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt</td>
</tr>
<tr>
<td>2016-11-23</td>
<td>8776899</td>
<td>elb/plaintext/2015/01/01/part-r-00001-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt</td>
</tr>
<tr>
<td>2016-11-23</td>
<td>9309800</td>
<td>elb/plaintext/2015/01/01/part-r-00002-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt</td>
</tr>
<tr>
<td>2016-11-23</td>
<td>9412570</td>
<td>elb/plaintext/2015/01/01/part-r-00003-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt</td>
</tr>
<tr>
<td>2016-11-23</td>
<td>10725938</td>
<td>elb/plaintext/2015/01/01/part-r-00004-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt</td>
</tr>
<tr>
<td>2016-11-23</td>
<td>9439710</td>
<td>elb/plaintext/2015/01/01/part-r-00005-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt</td>
</tr>
<tr>
<td>2016-11-23</td>
<td>0</td>
<td>elb/plaintext/2015/01/01_/folder$</td>
</tr>
<tr>
<td>2016-11-23</td>
<td>9012723</td>
<td>elb/plaintext/2015/01/02/part-r-00006-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt</td>
</tr>
<tr>
<td>2016-11-23</td>
<td>7571816</td>
<td>elb/plaintext/2015/01/02/part-r-00007-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt</td>
</tr>
<tr>
<td>2016-11-23</td>
<td>9673393</td>
<td>elb/plaintext/2015/01/02/part-r-00008-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt</td>
</tr>
<tr>
<td>2016-11-23</td>
<td>11979218</td>
<td>elb/plaintext/2015/01/02/part-r-00009-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt</td>
</tr>
<tr>
<td>2016-11-23</td>
<td>9546833</td>
<td>elb/plaintext/2015/01/02/part-r-00010-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt</td>
</tr>
<tr>
<td>2016-11-23</td>
<td>9546833</td>
<td>elb/plaintext/2015/01/02/part-r-00010-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt</td>
</tr>
</tbody>
</table>
Preparing Partitioned and Nonpartitioned Data for Querying
In this case, you would have to use ALTER TABLE ADD PARTITION to add each partition manually. 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. 505).

For example, to load the data in s3://athena-examples-myregion/elb/plaintext/2015/01/01/, you can run the following. 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.

```
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/
```

To remove a partition, use ALTER TABLE DROP PARTITION (p. 506).

**Additional Resources**

- 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. 186).
- You can automate adding partitions by using the JDBC driver (p. 87).

**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. 36). 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. For an example, see Amazon Kinesis Data Firehose Example (p. 132).

**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. 126).

**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.

**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. 121)
- Supported Types for Partition Projection (p. 126)
- Dynamic ID Partitioning (p. 130)
- Amazon Kinesis Data Firehose Example (p. 132)

## 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. 511) queries, or AWS Glue API operations. The following procedure shows how to set the properties in the AWS Glue console. For an example that uses a `CREATE TABLE` statement, see the Amazon Kinesis Data Firehose Example (p. 132).

**To configure and enable partition projection using the AWS Glue console**

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. 126).

6. Following the guidance in Supported Types for Partition Projection (p. 126), 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 2000 through 2016.
7. Add a key-value pair to enable partition projection. For **Key**, enter `projection.enabled`, and for its **Value**, enter `true`. 
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 2000 to 2016.
### Setting up Partition Projection

```sql
1. Select distinct year from flights_parquet
2. order by year ASC
```

Results

<table>
<thead>
<tr>
<th></th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2000</td>
</tr>
<tr>
<td>2</td>
<td>2001</td>
</tr>
<tr>
<td>3</td>
<td>2002</td>
</tr>
<tr>
<td>4</td>
<td>2003</td>
</tr>
<tr>
<td>5</td>
<td>2004</td>
</tr>
<tr>
<td>6</td>
<td>2005</td>
</tr>
<tr>
<td>7</td>
<td>2006</td>
</tr>
<tr>
<td>8</td>
<td>2007</td>
</tr>
<tr>
<td>9</td>
<td>2008</td>
</tr>
<tr>
<td>10</td>
<td>2009</td>
</tr>
<tr>
<td>11</td>
<td>2010</td>
</tr>
<tr>
<td>12</td>
<td>2011</td>
</tr>
<tr>
<td>13</td>
<td>2012</td>
</tr>
<tr>
<td>14</td>
<td>2013</td>
</tr>
<tr>
<td>15</td>
<td>2014</td>
</tr>
<tr>
<td>16</td>
<td>2015</td>
</tr>
<tr>
<td>17</td>
<td>2016</td>
</tr>
</tbody>
</table>
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 are missing projection configuration: [column_name] (table database_name.table_name).

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. 121), 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.

<table>
<thead>
<tr>
<th>Template 1</th>
<th>Template 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>s3://bucket/table_root/a=${a}/${b}/some_static_subdirectory/${c}/</td>
<td>s3://bucket/table_root/c=${c}/${b}/some_static_subdirectory/${a}/${b}/${c}/${c}/</td>
</tr>
</tbody>
</table>

   For the same table, the following example template value is invalid because it contains no placeholder for column c.

<table>
<thead>
<tr>
<th>Invalid Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>s3://bucket/table_root/a=${a}/${b}/some_static_subdirectory/</td>
</tr>
</tbody>
</table>

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:
Supported Types for Partition Projection

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Example Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>projection.columnName.type enum</td>
<td></td>
<td>Required. The projection type to use for column columnName. The value must be enum (case insensitive) to signal the use of the enum type. Leading and trailing white space is allowed.</td>
</tr>
<tr>
<td>projection.columnName.values</td>
<td>A,B,C,D,E,F,G,Unknown</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>0,10 -1,8675309 0001,9999</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...</td>
</tr>
<tr>
<td>Property Name</td>
<td>Example Values</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>successive partition values for the column columnName</strong>. 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. 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>
<tr>
<td><strong>projection.columnName.digits</strong></td>
<td>5</td>
<td>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 &quot;1,3&quot; that has a digits value of &quot;1&quot; produces the values 1, 2, and 3. The same range value with a digits value of &quot;2&quot; 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.</td>
</tr>
</tbody>
</table>

**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><strong>projection.columnName.type</strong></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>
<tr>
<td><strong>projection.columnName.range</strong></td>
<td>201701, 201812</td>
<td>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:</td>
</tr>
<tr>
<td>Property Name</td>
<td>Example Values</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>\s<em>NOW\s</em>{{[+-]}\s*{[0-9]+}\s*{YEARS?</td>
<td>MONTHS?</td>
<td>WEEKS?</td>
</tr>
<tr>
<td>projection.columnName.format</td>
<td>yyyyMM &lt;br&gt; dd-MM-yyyy &lt;br&gt; dd-MM-yyyy-HH-mm-ss</td>
<td>Required. A date format string based on the Java date format <code>DateTimeFormatter</code>. Can be any supported <code>java.time.*</code> type.</td>
</tr>
<tr>
<td>projection.columnName.interval</td>
<td>5</td>
<td>A positive integer that specifies the interval between successive partition values for column <code>columnName</code>. For example, a range value of 2017-01,2018-12 with an interval value of 1 and an <code>interval.unit</code> value of <code>MONTHS</code> produces the values 2017-01, 2017-02, 2017-03, and so on. The same range value with an <code>interval.value</code> of 2 and an <code>interval.unit</code> value of <code>MONTHS</code> produces the values 2017-01, 2017-03, 2017-05, and so on. Leading and trailing white space is allowed. When the provided dates are at single-day or single-month precision, the <code>interval</code> is optional and defaults to 1 day or 1 month, respectively. Otherwise, <code>interval</code> is required.</td>
</tr>
<tr>
<td>projection.columnName.interval.unit</td>
<td>YEARS &lt;br&gt; MONTHS &lt;br&gt; WEEKS &lt;br&gt; DAYS &lt;br&gt; HOURS &lt;br&gt; MINUTES &lt;br&gt; SECONDS &lt;br&gt; MILLISECONDS</td>
<td>A time unit word that represents the serialized form of a <code>ChronoUnit</code>. Possible values are <code>YEARS</code>, <code>MONTHS</code>, <code>WEEKS</code>, <code>DAYS</code>, <code>HOURS</code>, <code>MINUTES</code>, <code>SECONDS</code>, or <code>MILLISECONDS</code>. These values are case insensitive. When the provided dates are at single-day or single-month precision, the <code>interval.unit</code> is optional and defaults to 1 day or 1 month, respectively. Otherwise, the <code>interval.unit</code> is required.</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 on an injected column fail if a filter expression on the column allows multiple values.
• 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>projection.columnName.type</td>
<td>injected</td>
<td>Required. The projection type to use for the column columnName. 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. 130).

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.

**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 on writing your own user-defined function like the scalar bucketing function in the preceding example, see Querying with User Defined Functions (p. 262).

### Amazon Kinesis Data Firehose Example

Kinesis Data Firehose stores data in Amazon S3 in the following path format:

```
s3://bucket/folder/yyyy/MM/dd/HH/file.extension
```

Normally, to use Athena to query Kinesis Data Firehose data without using partition projection, you create a table for Kinesis Data Firehose logs in Athena. Then you must add partitions to your table in the AWS Glue Data Catalog every hour when Kinesis Data Firehose creates a partition.

By using partition projection, you can use a one-time configuration to inform Athena where the partitions reside. The following `CREATE TABLE` example assumes a start date of 2018-01-01 at midnight. Note the use of `NOW` for the upper boundary of the date range, which allows new data to automatically become queryable at the appropriate UTC time.

```
CREATE EXTERNAL TABLE my_table
(
  ...
)
PARTITIONED BY
(
  datehour STRING
)
LOCATION "s3://bucket/table-name/"
TBLPROPERTIES
(
  "projection.enabled" = "true",
  "projection.datehour.type" = "date",
  "projection.datehour.range" = "2018/01/01,0000,NOW",
  "projection.datehour.format" = "yyyy/MM/dd/HH",
  "projection.datehour.interval" = "1",
  "projection.datehour.interval.unit" = "HOURS",
  "storage.location.template" = "s3://bucket/table-name/${datehour}" )
```

Kinesis Data Firehose adds the partition prefix after `table-name` for you. In the Kinesis console, `table-name` appears in the Custom Prefix field.

With this table you can run queries like the following, without having to manually add partitions:

```
SELECT *
FROM my_table
WHERE datehour >= '2018/02/03/00'
AND datehour < '2018/02/03/04'
```
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. 133)
- Supported SerDes and Data Formats (p. 134)
- Compression Formats (p. 160)

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. 150)
- Querying Amazon CloudFront Logs (p. 274)
- Querying Amazon EMR Logs (p. 282)
- Querying Amazon VPC Flow Logs (p. 294)
- Using CTAS and INSERT INTO for ETL and Data Analysis (p. 186)

- 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'
WITH SERDEPROPERTIES (
  'serialization.format' = ',',
  'field.delim' = ',',
  'collection.delim' = '|',
  'mapkey.delim' = ':',
)
For examples of `ROW FORMAT SERDE`, see the following topics:

- Avro SerDe (p. 136)
- CloudTrail SerDe (p. 138)
- Grok SerDe (p. 143)
- JSON SerDe Libraries (p. 146)
- OpenCSVSerDe for Processing CSV (p. 140)
- Regex SerDe (p. 138)

**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. 177).

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>
</table>
| CSV (Comma-Separated Values) | For data in CSV, each line represents a data record, and each record consists of one or more fields, separated by commas. | • Use the LazySimpleSerDe for CSV, TSV, and Custom-Delimited Files (p. 150) 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. 140) when your data includes quotes in values or uses the UNIX numeric format for TIMESTAMP (for example, 1564610311). |
<table>
<thead>
<tr>
<th>Data Format</th>
<th>Description</th>
<th>SerDe types supported in Athena</th>
</tr>
</thead>
<tbody>
<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. 150) and specify the separator character as FIELDS TERMINATED BY <code>\t</code>.</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. 150) and specify a custom single-character delimiter.</td>
</tr>
</tbody>
</table>
| JSON (JavaScript Object Notation) | For JSON data, each line represents a data record, and each record consists of attribute-value pairs and arrays, separated by commas. | • Use the Hive JSON SerDe (p. 146).  
  • Use the OpenX JSON SerDe (p. 147). |
| Apache Avro              | A format for storing data in Hadoop that uses JSON-based schemas for record values. | Use the Avro SerDe (p. 136). |
| ORC (Optimized Row Columnar) | A format for optimized columnar storage of Hive data. | Use the ORC SerDe (p. 155) and ZLIB compression. |
| Apache Parquet           | A format for columnar storage of data in Hadoop.                            | Use the Parquet SerDe (p. 157) and SNAPPY compression. |
| Logstash logs            | A format for storing logs in Logstash.                                      | Use the Grok SerDe (p. 143). |
| Apache WebServer logs    | A format for storing logs in Apache WebServer.                              | Use the Grok SerDe (p. 143) or Regex SerDe (p. 138). |
| CloudTrail logs          | A format for storing logs in CloudTrail.                                    | • Use the CloudTrail SerDe (p. 138) to query most fields in CloudTrail logs.  
  • Use the OpenX JSON SerDe (p. 147) for a few fields where their format depends on the service.  
    For more information, see CloudTrail SerDe (p. 138). |

**Topics**

- Avro SerDe (p. 136)
- Regex SerDe (p. 138)
- CloudTrail SerDe (p. 138)
- OpenCSVserDe for Processing CSV (p. 140)
- Grok SerDe (p. 143)
- JSON SerDe Libraries (p. 146)
- LazySimpleSerDe for CSV, TSV, and Custom-Delimited Files (p. 150)
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 -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",
  "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.

`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;`

**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 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/).

```
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(\[\^/\]+)\[/\]\\])]\.*\%20(\[\^/\]+)\[/\]\\]);"
    LOCATION 's3://athena-examples-myregion/cloudfront/plaintext/';
```

CloudTrail SerDe

AWS CloudTrail is a service that records AWS API calls and events for Amazon Web Services accounts. CloudTrail generates encrypted logs and stores them in Amazon S3. You can use Athena to query these logs directly from Amazon S3, specifying the LOCATION of logs.

To query CloudTrail logs in Athena, create table from the logs and use the CloudTrail SerDe to deserialize the logs data.

In addition to using the CloudTrail SerDe, instances exist where you need to use a different SerDe or to extract data from JSON. Certain fields in CloudTrail logs are STRING values that may have a variable data format, which depends on the service. As a result, the CloudTrail SerDe is unable to predictably deserialize them. To query the following fields, identify the data pattern and then use a different
SerDe, such as the OpenX JSON SerDe (p. 147). Alternatively, to get data out of these fields, use JSON\_EXTRACT functions. For more information, see Extracting Data From JSON (p. 255).

- requestParameters
- responseElements
- additionalEventData
- serviceEventDetails

SerDe Name
CloudTrail SerDe

Library Name
com.amazon.emr.hive.serde.CloudTrailSerde

Examples

The following example uses the CloudTrail SerDe on a fictional set of logs to create a table based on them.

In this example, the fields requestParameters, responseElements, and additionalEventData are included as part of STRUCT data type used in JSON. To get data out of these fields, use JSON\_EXTRACT functions. For more information, see Extracting Data From JSON (p. 255).

```
CREATE EXTERNAL TABLE cloudtrail_logs (  
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>>,  
    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,  
    resources ARRAY<STRUCT<  
        ARN:STRING,  
        ...>>>
```
The following query returns the logins that occurred over a 24-hour period:

```sql
SELECT
  useridentity.username,
  sourceipaddress,
  eventtime,
  additionaleventdata
FROM default.cloudtrail_logs
WHERE eventname = 'ConsoleLogin'
  AND eventtime >= '2017-02-17T00:00:00Z'
  AND eventtime < '2017-02-18T00:00:00Z';
```

For more information, see Querying AWS CloudTrail Logs (p. 276).

### OpenCSVSerDe for Processing CSV

When you create an Athena table for CSV data, determine the SerDe to use what types of values it 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. 150).
- 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:

  ```sql
  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.fffffffff" (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 (p. 172) over the table and use CAST to convert to the desired type.

Example 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.

```sql
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:

```sql
SELECT * FROM testtimestamp1
```

The query returns the following result, showing the date and time data:

<table>
<thead>
<tr>
<th>profile_id</th>
<th>creationdate</th>
<th>creationdatetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>unixvalue</td>
<td>18276</td>
<td>1579146280000</td>
</tr>
<tr>
<td>2020-01-15</td>
<td>03:44:40.000</td>
<td>2020-01-15 03:44:40.000</td>
</tr>
</tbody>
</table>

Example Example: Escaping \t or \n
Consider the following test data:

" \t\t\n 123 \t\t\n ",abc
The following statement creates a table in Athena, specifying that "escapeChar" = "\".

```
CREATE EXTERNAL TABLE test1 (
    f1 string,
    s2 string)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES ("separatorChar" = ",", "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:
```
f1            s2
\t\n 123 \t\n            abc
456                          xyz
```

### 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. 294) and [Querying Amazon CloudFront Logs](p. 274).

### 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"  
"a","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:

```
col1 col2 col3 col4
------- ------ ------ ------
a1    a2    a3    a4
1      2     abc   def
a      a1    abc3  ab4
```

**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 ([^"\]*\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:amqListenerContainerPool23[P:AJ|ABCDE9614B6F5E49||
2017-09-12T12:10:11.172-0700],
executionTime=7290, tenantId=12456, userId=123123f8535f8d76015374e7a1d87c3c,
shard=testapp1,
jobId=12312345e5e7d0015e777fb2e03f3c, 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.

```
CREATE EXTERNAL TABLE bltest (
  timestamp STRING,
  loglevel STRING,
  processtype STRING,
  processid STRING,
```
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 shown in this snippet from the example query: 'input.grokCustomPatterns'='INSIDE_QS (^[^"\"]*)\nINSIDE_BRACKETS ([^\[]*])'.

```
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}]%{INSERT:remote_ip} %{NOTSPACE:requester} %{NOTSPACE:request_id} %{NOTSPACE:operation} %{NOTSPACE:key} %{INSERT:request_uri} %{NOTSPACE:http_status} %{NOTSPACE:error_code} %{NOTSPACE:bytes_sent} %{NOTSPACE:object_size} %{NOTSPACE:total_time} %{NOTSPACE:turnaround_time} %{INSERT:referrer} %{INSERT:user_agent} %{NOTSPACE:version_id}',
    'input.grokCustomPatterns']='INSIDE_QS (^[^"\"]*)\nINSIDE_BRACKETS ([^\[]*])')
STORED AS INPUTFORMAT
  'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT
  'org.apache.hadoop.mapred.TextOutputFormat'
LOCATION 's3://mybucket/samples/';
```
JSON SerDe Libraries

In Athena, you can use two 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. 146)
- The OpenX JSON SerDe (p. 147)

SerDe Names

Hive-JsonSerDe

Openx-JsonSerDe

Library Names

Use one of the following:

org.apache.hive.hcatalog.data.JsonSerDe

org.openx.data.jsonserde.JsonSerDe

Hive JSON SerDe

The Hive JSON SerDe is commonly used to process JSON data like events. These events are represented as blocks of JSON-encoded text separated by a new line. The Hive JSON SerDe does not allow duplicate keys in map or struct key names.

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).

```
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'
WITH SERDEPROPERTIES ( 'paths'='requestbegintime, adid, impressionid, referrer, useragent, usercookie, ip' )
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 use the timestamp.formats parameter in SERDEPROPERTIES to specify a comma-separated list of one or more timestamp patterns, as in the following example:

```
... 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. 520) to load the table and make it queryable from Athena:

```
MSCK REPAIR TABLE impressions
```

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")
```

If you have two keys like URL and Url that are the same when they are in lowercase, an error like the following can occur:
HIVE_CURSOR_ERROR: Row is not a valid JSON Object - JSONException: Duplicate key "url"

To resolve this, set the case.insensitive property to FALSE and map the keys to different names, as in the following example:

```sql
ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'
WITH SERDEPROPERTIES ("case.insensitive" = "FALSE", "mapping.url1" = "URL",
"mapping.url2" = "Url")
```

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. 107). For example, if you have a JSON key named timestamp, use the following syntax to map the key to a column named ts:

```sql
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.

```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.openx.data.jsonserde.JsonSerDe'
with serdeproperties ( "paths"='requestbegintime, adid, impressionid, referrer, useragent,  
usercookie, ip' )
LOCATION 's3://myregion.elasticmapreduce/samples/hive-ads/tables/impressions';
```

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",
  "User": {
    "Id": 1234,
    "Username": "bob1234",
    "Name": "Bob",
    "ShippingAddress": {
      "Address1": "123 Main St.",
      "Address2": null,
      "City": "Seattle",
      "State": "WA"
    },
    "Orders": [
      {
        "ItemId": 6789,
        "OrderDate": "11/11/2017"
      },
      {
        "ItemId": 4352,
        "OrderDate": "12/12/2017"
      }
    ]
  }
}
```

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.

```sql
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)
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.

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. 294) and Querying Amazon CloudFront Logs (p. 274).

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. 150)
- TSV Example (p. 153)

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,
airlneid INT,
carrier STRING,
tailnum STRING,
flightnum STRING,
originalairportid INT,
originalportseqid INT,
originmarketid INT,
origin STRING,
originname STRING,
originstate STRING,
originstatefips STRING,
originstatename STRING,
originwac INT,
destairportid INT,
destportseqid INT,
destmarketid INT,
dest STRING,
destname STRING,
deststate STRING,
deststatefips STRING,
deststatename STRING,
destwac INT,
crsdeptime STRING,
deptime STRING,
depdelay INT,
depdelayminutes INT,
depdelll INT,
departuredelaygroups INT,
deptimebklk STRING,
taxiout INT,
wheeloff STRING,
wheelon STRING,
taxiin INT,
crsarrtime INT,
arrtime STRING,
arrdelay INT,
arrdelayminutes INT,
arrdelll INT,
arrivaldelaygroups INT,
arrtimebklk STRING,
cancelled INT,
cancellationcode STRING,
diverted INT,
crselapsedtime 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, 
divairport STRING, 
divairportid INT, 
divairportseqid INT, 
div1wheelson STRING, 
div1totaltime INT, 
div1longestgtime INT, 
div1wheelsoff STRING, 
div1tailnum STRING, 
div2airport STRING, 
div2airportid INT, 
div2airportseqid INT, 
div2wheelson STRING, 
div2totaltime INT, 
div2longestgtime INT, 
div2wheelsoff STRING, 
div2tailnum STRING, 
div3airport STRING, 
div3airportid INT, 
div3airportseqid INT, 
div3wheelson STRING, 
div3totaltime INT, 
div3longestgtime INT, 
div3wheelsoff STRING, 
div3tailnum STRING, 
div4airport STRING, 
div4airportid INT, 
div4airportseqid INT, 
div4wheelson STRING, 
div4totaltime INT, 
div4longestgtime INT, 
div4wheelsoff STRING, 
div4tailnum STRING, 
div5airport STRING, 
div5airportid INT, 
div5airportseqid INT, 
div5wheelson STRING, 
div5totaltime 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, 
  originickname STRING, 
  originstate STRING, 
  originstatefips STRING, 
  originstatename STRING, 
  originwac INT, 
  destairportid INT, 
  destairportseqid INT, 
  destcitymarketid INT, 
  dest STRING, 
  destickname STRING, 
  deststate STRING, 
  deststatefips STRING, 
  deststatename STRING, 
  destwac INT, 
  crsdeptime STRING, 
  deptime STRING, 
  depdelay INT, 
  depdelayminutes INT, 
  depdelay15 INT, 
  departuredelaygroups INT, 
  depdelaytimeblk STRING, 
  taxiout INT, 
  wheelsoff STRING, 
  wheelson STRING, 
  taxiin INT, 
  crsarrtime INT, 
  arrtime STRING, 
  arrdelay INT, 
  arrdelayminutes INT, 
  arrdelay15 INT, 
  arrivaldelaygroups INT, 
  arrtimeblk STRING, 
  
```
cancelled INT,
cancellationcode STRING,
diverted INT,
crselepapsedtime INT,
actualelepapsedtime INT,
airtime INT,
flights INT,
distance INT,
distancegroup INT,
carrierdelay INT,
weatherdelay INT,
nasdelay INT,
securitydelay INT,
lateaircraftdelay INT,
firstdepetime STRING,
totaladdgtime INT,
longestaddgtime INT,
divairportlandings INT,
divreacheddest INT,
divactualelepapsedtime INT,
divarrdelay 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.

### ORC SerDe

#### SerDe Name
OrcSerDe

#### Library Name
This is the SerDe class for data in the ORC format. It passes the object from ORC to the reader and from ORC to the writer: OrcSerDe

#### 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,
airlineid INT,
carrier STRING,
tailnum STRING,
flightnum STRING,
originairportid INT,
originairportseqid INT,
originairport STRING,  
originairportspecid INT,
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
originairportid INT,  
originairportseqid INT,
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,
depdelay INT,
depdelayminutes INT,
depdell15 INT,
departuredelaygroups INT,
depertime STRING,
taxiout INT,
wheelsoff STRING,
wheelson STRING,
taxiin INT,
crsarrtime INT,
arrtime STRING,
arrdelay INT,
arrdelayminutes INT,
arrdell15 INT,
arrivaldelaygroups INT,
arrtimeblk STRING,
cancelled INT,
cancellationcode STRING,
diverted INT,
actuelapsedtime INT,
arr elapsedtime INT,
cr selapsed time INT,
arrelapsedtime INT,
airtime INT,
flights INT,
distance INT,
distancegroup INT,
carriedelay INT,
weatherdelay INT,
nasdelay INT,
securitydelay INT,
lateaircraftdelay INT,
firstdeptime STRING,
totaladdgtime INT,
longestaddgtime INT,
divairportlandings INT,
divreachdest INT,
divactualelapsedtime INT,
divar delay INT,
divdistance INT,
divairport STRING,
divairportid INT,
divairportseqid INT,
divwheelson STRING,
divtotaltime INT,
divlongesttime INT,
divwheelsoff STRING,
divtailnum STRING,
divaairport STRING,
divaairportid INT,
div2airportseqid INT,
div2wheelson STRING,
div2totalgtime INT,
div2longestgtime INT,
div2wheelsoff 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)
STORED AS ORC
LOCATION 's3://athena-examples-myregion/flight/orc/

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. 514) queries. For more information, see Creating a Table from Query Results (CTAS) (p. 177), Examples of CTAS Queries (p. 183) and Using CTAS and INSERT INTO for ETL and Data Analysis (p. 186).
Library Name

Athena uses this class when it needs to deserialize data stored in Parquet:

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,
    originalairportseqid INT,
    originairportid INT,
    originairportscreenid INT,
    originairportid INT,
    originairportid INT,
    originairportid INT,
    originairportid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
    destairportseqid INT,
    destairportid INT,
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,
totaladdgttime INT,
longestaddgttime INT,
divairportlandings INT,
divreacheddest INT,
divactualelapsedtime INT,
divarrdelay INT,
divdistance INT,
divairport STRING,
divairportid INT,
divairportseqid INT,
divwheelson STRING,
divtotalgttime INT,
divlongestgttime INT,
divwheelsoff STRING,
divtailnum STRING,
div2airport STRING,
div2airportid INT,
div2airportseqid INT,
div2wheelson STRING,
div2totalgttime INT,
div2longestgttime INT,
div2wheelsoff STRING,
div2tailnum STRING,
div3airport STRING,
div3airportid INT,
div3airportseqid INT,
div3wheelson STRING,
div3totalgttime INT,
div3longestgttime INT,
div3wheelsoff STRING,
div3tailnum STRING,
div4airport STRING,
div4airportid INT,
div4airportseqid INT,
div4wheelson STRING,
div4totalgttime INT,
div4longestgttime INT,
div4wheelsoff STRING,
div4tailnum STRING,
div5airport STRING,
div5airportid INT,
div5airportseqid INT,
div5wheelson STRING,
div5totalgttime INT,
div5longestgttime INT,
div5wheelsoff STRING,
div5tailnum STRING
)
PARTITIONED BY (year STRING)
STORED AS PARQUET
LOCATION 's3://athena-examplea-myregion/flight/parquet/'
Compression Formats

The compression formats listed in this section are used for CREATE TABLE (p. 511) queries. For CTAS queries, Athena supports GZIP and SNAPPY (for data stored in Parquet and ORC). If you omit a format, GZIP is used by default. For more information, see CREATE TABLE AS (p. 514).

Athena supports the following compression formats:

- **SNAPPY** – The default compression format for files in the Parquet data storage format.
- **ZLIB** – The default compression format for files in the ORC data storage format.
- **LZO** – Format that uses the Lempel–Ziv–Oberhumer algorithm.
- **GZIP** – Athena can query data in this format directly if the data files have the .gz extension. No special directive is required in the CREATE TABLE statement.
- **BZIP2** – Format that uses the Burrows-Wheeler algorithm.

**Note**

In rare cases, a known issue in Athena engine version 1 can cause records to be silently dropped when the BZIP2 format is used. For this reason, use of the BZIP2 format in Athena engine version 1 is not recommended.

Specifying Compression Formats

To specify a compression format for the Parquet SerDe (p. 157) or ORC SerDe (p. 155) in a CREATE TABLE statement, use the TBLPROPERTIES (p. 508) clause. To specify a compression format for Parquet or ORC in a CTAS (p. 177) statement, use the WITH clause. For more information, see CTAS Table Properties (p. 515) and Example: Specifying Data Storage and Compression Formats (p. 184).

Notes and Resources

- 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 on using compression, see section 3 ("Compress and split files") of the AWS Big Data Blog post Top 10 Performance Tuning Tips for Amazon Athena.
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. 18). 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.

Topics

- Working with Query Results, Output Files, and Query History (p. 162)
- Working with Views (p. 172)
- Creating a Table from Query Results (CTAS) (p. 177)
- Querying with Prepared Statements (p. 195)
- Handling Schema Updates (p. 197)
- Querying Arrays (p. 206)
- Querying Geospatial Data (p. 223)
- Using Athena to Query Apache Hudi Datasets (p. 248)
- Querying JSON (p. 253)
- Using Machine Learning (ML) with Amazon Athena (p. 260)
- Querying with User Defined Functions (p. 262)
- Querying AWS Service Logs (p. 270)
- Querying AWS Glue Data Catalog (p. 303)
- Querying Web Server Logs Stored in Amazon S3 (p. 307)

For considerations and limitations, see Considerations and Limitations for SQL Queries in Amazon Athena (p. 528).

Working with Query Results, Output Files, and Query History

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.
Getting a Query ID

Each query that runs is known as a *query execution*. The query execution has a unique identifier known as the query ID or query execution ID. To work with query result files, and to quickly find query result files, you need the query ID. We refer to the query ID in this topic as *QueryID*.

To use the Athena console to get the *QueryID* of a query that ran

1. Choose **History** from the navigation bar.

2. From the list of queries, choose the query status under **State**—for example, **Succeeded**. The query ID shows in a pointer tip.
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. 167) 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><strong>QueryID.csv</strong></td>
<td>DML query results files are saved in comma-separated values (CSV) format.</td>
</tr>
<tr>
<td></td>
<td><strong>QueryID.txt</strong></td>
<td>DDL query results are saved as plain text files.</td>
</tr>
</tbody>
</table>
### 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>Query metadata files</td>
<td>QueryID.csv.metadata</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></td>
<td>QueryID.txt.metadata</td>
<td></td>
</tr>
<tr>
<td>Data manifest files</td>
<td>QueryID-manifest.csv</td>
<td>Data manifest files are generated to track files that Athena creates in Amazon S3 data source locations when an INSERT INTO (p. 479) 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
        },
        "Statistics": {
            "DataScannedInBytes": 5944497,
        }
    }
}
```
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, or using the query History.

**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 download the query results file of the most recent query

1. Enter your query in the query editor and then choose Run query.

When the query finishes running, the Results pane shows the query results.

2. To download the query results file, choose the file icon in the query results pane. Depending on your browser and browser configuration, you may need to confirm the download.
To download a query results file for an earlier query

1. Choose History.
2. Page through the list of queries until you find the query, and then, under Action for the query, choose Download results.

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. If no query results location is specified, the query fails with an error.

To specify a client-side setting query result location using the Athena console

1. Switch (p. 434) 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. For Query result location and encryption, do one of the following:
   - Choose Select, and then choose a path to an existing Amazon S3 folder.
   - Enter a path to an Amazon S3 folder, including the trailing slash. If the bucket or folder does not exist, Athena creates it for you when you first run a query.

   **Note**
   Be sure to include the trailing slash. Failing to include the trailing slash results in the error Invalid S3 folder location.

The Amazon S3 location that you enter is used for subsequent queries. You can change this location later if you want.

If you are a member of a workgroup that specifies a query result location and overrides client-side settings, the option to change the query result location is unavailable, as the following image shows:
Previously Created Default Locations

Previously, 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. Choose **Workgroup**: `CurrentWorkgroupName` in the navigation bar.
2. Do one of the following:
   - If editing an existing workgroup, select it from the list, choose **View details**, and then choose **Edit Workgroup**.
   - If creating a new workgroup, choose **Create workgroup**.
3. For **Query result location**, choose the **Select** folder.
4. From the list of S3 locations, choose the blue arrow successively until the bucket and folder you want to use appears in the top line. Choose **Select**.
5. Under **Settings**, do one of the following:
• Select **Override client-side settings** to save query files in the location that you specified above for all queries that members of this workgroup run.

• Clear **Override client-side settings** to save query files in the location that you specified above have the query location that you specified above only when workgroup members run queries using the Athena API, ODBC driver, or JDBC driver without specifying an output location in Amazon S3.

6. If editing a workgroup, choose **Save**. If creating a workgroup, choose **Create workgroup**.

**Viewing Query History**

You can use the Athena console to see the queries that succeeded and failed, download query result files for the queries that succeeded, and view error details for the queries that failed. Athena keeps a query history for 45 days.

**To view query history in the Athena console**

2. Choose the **History** tab. The **History** tab shows information about each query that ran.

3. Do one of the following:

   • To see a query statement in the Query Editor, choose the text of the query in the **Query** column. Longer query statements are abbreviated.
• To see a query ID, choose its State (Succeeded, Failed, or Cancelled). The query ID shows in a pointer tip.

• To download the results of a successful query into a .csv file, choose Download results.

• To see the details for a query that failed, choose Error details for the query.

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.

Working with Views

A view in Amazon Athena is a logical, not a physical table. The query that defines a view runs each time the view is referenced in a query.

You can create a view from a SELECT query and then reference this view in future queries. For more information, see CREATE VIEW (p. 516).

Topics

- When to Use Views? (p. 172)
- Supported Actions for Views in Athena (p. 173)
- Considerations for Views (p. 173)
- Limitations for Views (p. 174)
- Working with Views in the Console (p. 174)
- Creating Views (p. 175)
- Examples of Views (p. 176)
- Updating Views (p. 177)
- Deleting Views (p. 177)

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. 516)</td>
<td>Creates a new view from a specified SELECT query. For more information, see Creating Views (p. 175).</td>
</tr>
<tr>
<td></td>
<td>The optional OR REPLACE clause lets you update the existing view by replacing it.</td>
</tr>
<tr>
<td>DESCRIBE VIEW (p. 518)</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. 519)</td>
<td>Deletes an existing view. The optional IF EXISTS clause suppresses the error if the view does not exist. For more information, see Deleting Views (p. 177).</td>
</tr>
<tr>
<td>SHOW CREATE VIEW (p. 523)</td>
<td>Shows the SQL statement that creates the specified view.</td>
</tr>
<tr>
<td>SHOW VIEWS (p. 527)</td>
<td>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. You can also see the list of views in the left pane in the console.</td>
</tr>
<tr>
<td>SHOW COLUMNS (p. 522)</td>
<td>Lists the columns in the schema for a view.</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. 335) 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. 106).

• 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. 107).

• You cannot use views with federated data sources, external Hive metastores, or UDFs.

• 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. 331).

Working with Views in the Console

In the Athena console, you can:

• Locate all views in the left pane, where tables are listed. Athena runs a SHOW VIEWS (p. 527) operation to present this list to you.

• Filter views.

• Preview a view, show its properties, edit it, or delete it.

To list the view actions in the console

A view shows up in the console only if you have already created it.

1. In the Athena console, choose Views, choose a view, then expand it.

   The view displays, with the columns it contains, as shown in the following example:

   ![salary_view](image)

   • id (string)
   • name (string)

2. In the list of views, choose a view, and open the context (right-click) menu. The actions menu icon (•) is highlighted for the view that you chose, and the list of actions opens, as shown in the following example:
Creating Views

You can create a view from any SELECT query.

**To create a view in the console**

Before you create a view, choose a database and then choose a table. Run a SELECT query on a table and then create a view from it.

1. In the Athena console, choose **Create view**.

   ![Create view](image)

   In the Query Editor, a sample view query displays.

2. Edit the sample view query. Specify the table name and add other syntax. For more information, see CREATE VIEW (p. 516) and Examples of Views (p. 176).

   View names cannot contain special characters, other than underscore (_). See Names for Tables, Databases, and Columns (p. 106). Avoid using Reserved Keywords (p. 107) for naming views.

3. Run the view query, debug it if needed, and save it.

   Alternatively, create a query in the Query Editor, and then use **Create view from query**.
If you run a view that is not valid, Athena displays an error message.

If you delete a table from which the view was created, when you 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. 523).

**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`:

```
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.

```
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;
```
Updating Views

After you create a view, it appears in the Views list in the left pane.

To edit the view, choose it, choose the context (right-click) menu, and then choose Show/edit query. You can also edit the view in the Query Editor. For more information, see CREATE VIEW (p. 516).

Deleting Views

To delete a view, choose it, choose the context (right-click) menu, and then choose Delete view. For more information, see DROP VIEW (p. 519).

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. 514).

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 into other storage formats, such as Parquet and ORC. This improves query performance and reduces query costs in Athena. For information, see Columnar Storage Formats (p. 110).
- Create copies of existing tables that contain only the data you need.

Topics

- Considerations and Limitations for CTAS Queries (p. 177)
- Running CTAS Queries in the Console (p. 179)
- Bucketing vs Partitioning (p. 182)
- Examples of CTAS Queries (p. 183)
- Using CTAS and INSERT INTO for ETL and Data Analysis (p. 186)
- Using CTAS and INSERT INTO to Create a Table with More Than 100 Partitions (p. 192)

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. 514).</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. 107), 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. 106).</td>
</tr>
<tr>
<td>Item</td>
<td>What You Need to Know</td>
</tr>
<tr>
<td>------</td>
<td>-----------------------</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. 429) for query results location, Athena creates your table in the location <code>s3://&lt;workgroup-query-results-location&gt;/tables/&lt;query-id&gt;/</code>. To see the query results location specified for the workgroup, view the workgroup's details (p. 433). If your workgroup does not override the query results location, you can use the syntax <code>WITH (external_location = 's3://location/')</code> in your CTAS query to specify where your CTAS query results are stored. <strong>Note</strong> The <code>external_location</code> 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. If you omit the <code>external_location</code> syntax and are not using the workgroup setting, Athena uses your client-side setting (p. 168) for the query results location and creates your table in the location <code>s3://&lt;client-query-results-location&gt;/&lt;Unsaved-or-query-name&gt;/&lt;year&gt;/&lt;month/&lt;date&gt;/tables/&lt;query-id&gt;/</code>.</td>
</tr>
<tr>
<td>Locating Orphaned Files</td>
<td>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. 164) and DataManifestLocation.</td>
</tr>
<tr>
<td>Formats for storing query results</td>
<td>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. 184).</td>
</tr>
<tr>
<td>Compression formats</td>
<td>GZIP compression is used for CTAS query results by default. For Parquet and ORC, you can also specify SNAPPY. See Example: Specifying Data Storage and Compression Formats (p. 184).</td>
</tr>
<tr>
<td>Partition and Bucket Limits</td>
<td>You can partition and bucket the results data of a CTAS query. For more information, see Bucketing vs Partitioning (p. 182). 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 <code>WITH</code> clause that specifies properties of the destination table. For more information, see Example: Creating Bucketed and Partitioned Tables (p. 186) and Bucketing vs Partitioning (p. 182). 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. 192).</td>
</tr>
</tbody>
</table>
### Running CTAS Queries in the Console

In the Athena console, you can:

- Create a CTAS query from another query (p. 179)
- Create a CTAS query from scratch (p. 179)

#### To create a CTAS query from another query

1. Run the query, choose **Create**, and then choose **Create table from query**.

2. In the **Create a new table on the results of a query** form, complete the fields as follows:
   
   a. For **Database**, select the database in which your query ran.
   
   b. For **Table name**, specify the name for your new table. Use only lowercase and underscores, such as `my_select_query_parquet`.
   
   c. For **Description**, optionally add a comment to describe your query.
   
   d. For **Output location**, optionally specify the location in Amazon S3, such as `s3://my_athena_results/mybucket/`. If you don't specify a location and your workgroup does not **Override Client-Side Settings** (p. 429), the following predefined location is used: `s3://aws-athena-query-results-<account>-<region>/<query-name-or-unsaved>/year/month/date/<query-id>/`.
   
   e. For **Output data format**, select from the list of supported formats. Parquet is used if you don't specify a format. See **Columnar Storage Formats** (p. 110).
f. Choose **Next** to review your query and revise it as needed. For query syntax, see **CREATE TABLE AS (p. 514)**. The preview window opens, as shown in the following example:
Running CTAS Queries in the Console

To create a CTAS query from scratch

Use the CREATE TABLE AS SELECT template to create a CTAS query from scratch.

1. In the Athena console, choose Create table, and then choose CREATE TABLE AS SELECT.

2. In the Query Editor, edit the query as needed, For query syntax, see CREATE TABLE AS (p. 514).

3. Choose Run query.

4. Optionally, choose Save as to save the query.
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. 177).

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. 192).

  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. 514).

  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 CREATE TABLE AS (p. 514).

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

See also Examples of CTAS Queries (p. 183).
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

**Examples of CTAS Queries**

Use the following examples to create CTAS queries. For information about the CTAS syntax, see [CREATE TABLE AS (p. 514)](https://docs.aws.amazon.com/athena/latest/userguide/ctas-syntax.html).

In this section:

- Example: Duplicating a Table by Selecting All Columns (p. 183)
- Example: Selecting Specific Columns From One or More Tables (p. 184)
- Example: Creating an Empty Copy of an Existing Table (p. 184)
- Example: Specifying Data Storage and Compression Formats (p. 184)
- Example: Writing Query Results to a Different Format (p. 184)
- Example: Creating Unpartitioned Tables (p. 185)
- Example: Creating Partitioned Tables (p. 185)
- Example: Creating Bucketed and Partitioned Tables (p. 186)

**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
```

```sql
SELECT *
FROM old_table;
```
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:

```
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:

```
CREATE TABLE new_table AS
SELECT column_1, column_2, ...
FROM old_table_1, old_table_2, ...
```

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:

```
CREATE TABLE new_table
AS SELECT *
FROM old_table
WITH NO DATA;
```

Example Example: Specifying Data Storage and Compression Formats

The following example uses a CTAS query to create a new table with Parquet data from a source table in a different format. You can specify PARQUET, ORC, AVRO, JSON, and TEXTFILE in a similar way.

This example also specifies compression as SNAPPY. If omitted, GZIP is used. GZIP and SNAPPY are the supported compression formats for CTAS query results stored in Parquet and ORC.

```
CREATE TABLE new_table
WITH (format = 'Parquet',
     parquet_compression = 'SNAPPY')
AS SELECT *
FROM old_table;
```

The following example is similar, but it stores the CTAS query results in ORC and uses the orc_compression parameter to specify the compression format. If you omit the compression format, Athena uses GZIP by default.

```
CREATE TABLE new_table
WITH (format = 'ORC',
     orc_compression = 'SNAPPY')
AS SELECT *
FROM old_table;
```

Example 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:
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 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:

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:

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:

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:

CREATE TABLE ctas_orc_unpartitioned
WITH (format = 'ORC')
AS SELECT key1, name1, comment1 FROM table1;

In the following example, the format is Avro:

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. 515). For more information about choosing the columns for partitioning, see Bucketing vs Partitioning (p. 182).

**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/',
     partitioned_by = ARRAY['key1'])
AS SELECT name1, address1, comment1, key1
FROM tables1;
```

```sql
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. 515).

For information about choosing the columns for bucketing, see Bucketing vs Partitioning (p. 182).

```sql
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;
```

**Using CTAS and INSERT INTO for ETL and Data Analysis**

You can use Create Table as Select (CTAS (p. 177)) and INSERT INTO (p. 479) 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. 474) 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. 187)
- Step 2: Use CTAS to Partition, Convert, and Compress the Data (p. 188)
- Step 3: Use INSERT INTO to Add Data (p. 189)
- Step 4: Measure Performance and Cost Differences (p. 190)

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</th>
<th>s3://aws-bigdata-blog/artifacts/athena-ctas-insert-into-blog/</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total objects</td>
<td>41727</td>
</tr>
<tr>
<td>Size of CSV dataset</td>
<td>11.3 GB</td>
</tr>
<tr>
<td>Region</td>
<td>us-east-1</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.

<table>
<thead>
<tr>
<th>Date</th>
<th>Size</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019-10-31</td>
<td>413.1 KiB</td>
<td>artifacts/athena-ctas-insert-into-blog/2010.csv0000</td>
</tr>
<tr>
<td>2019-10-31</td>
<td>412.0 KiB</td>
<td>artifacts/athena-ctas-insert-into-blog/2010.csv0001</td>
</tr>
<tr>
<td>2019-10-31</td>
<td>34.4 KiB</td>
<td>artifacts/athena-ctas-insert-into-blog/2010.csv0002</td>
</tr>
<tr>
<td>2019-10-31</td>
<td>412.2 KiB</td>
<td>artifacts/athena-ctas-insert-into-blog/2010.csv0100</td>
</tr>
<tr>
<td>2019-10-31</td>
<td>412.7 KiB</td>
<td>artifacts/athena-ctas-insert-into-blog/2010.csv0101</td>
</tr>
</tbody>
</table>

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. 509) command to create a database.

CREATE DATABASE blogdb

3. Run the following statement to create a table (p. 511).

```sql
CREATE EXTERNAL TABLE `blogdb`.`original_csv`
(`id` string,
`date` string,
`element` string,
`datavalue` bigint,
`mflag` string,
...)
Step 2: Use CTAS to Partition, Convert, and Compress the Data

After you create a table, you can use a single CTAS (p. 177) 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.

```sql
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,
       element,
       datavalue,
       mflag,
       qflag,
       sflag,
       obsstime,
       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 (/).

```bash
aws s3 ls s3://your-bucket/optimized-data/
```
The output shows the partitions.

<table>
<thead>
<tr>
<th>Year</th>
<th>Size</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015</td>
<td>7.3 MiB</td>
<td>optimized-data/1201031_215021_00001_3f42d_1be48df2-3154-438b-b61d-8fb23809679d</td>
</tr>
<tr>
<td>2016</td>
<td>7.0 MiB</td>
<td>optimized-data/1201031_215021_00001_3f42d_2a57f4e2-ffa0-4be3-9c3f-28b16d86d5a</td>
</tr>
<tr>
<td>2017</td>
<td>9.9 MiB</td>
<td>optimized-data/1201031_215021_00001_3f42d_34381db1-00ca-4092-bd65-ab04e06dc799</td>
</tr>
<tr>
<td>2018</td>
<td>7.5 MiB</td>
<td>optimized-data/1201031_215021_00001_3f42d_354a2bc1-345f-499e-9073-096cb63308d</td>
</tr>
<tr>
<td>2019</td>
<td>6.9 MiB</td>
<td>optimized-data/1201031_215021_00001_3f42d_42da4cfd-6e21-40a1-8152-0b902da385a</td>
</tr>
</tbody>
</table>

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-28b16d86d5a
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-4996-9073-096cb63308d
2019-10-31 14:51:05 6.9 MiB optimized-data/year=2015/20191031_215021_00001_3f42d_42da4cfd-6e21-40a1-8152-0b902da385a
```

**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. 479) 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.

<table>
<thead>
<tr>
<th>Year</th>
<th>Size</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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_00003_nja5r_fd9906b7-06cf-4055-a05b-f050e139946e
Total Objects: 300
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,
    datavalue,
    mflag,
    qflag,
    sflag,
    obstime,
    substr("date",1,4) AS year
FROM original_csv
WHERE cast(substr("date",1,4) AS bigint) = 2020
```

**Note**
The `INSERT INTO` statement supports writing a maximum of 100 partitions to the destination table. However, to add more than 100 partitions, you can run multiple `INSERT INTO` statements. For more information, see Using CTAS and INSERT INTO to Create a Table with More Than 100 Partitions (p. 192).

### 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](https://aws.amazon.com/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') AND substr("date",1,4) = '2018'
   GROUP BY 1
   ```

6. Note the time that the query ran and the amount of data scanned.

7. Run the same query on the new table, noting the query runtime and amount of data scanned.

   ```sql
   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
   ```

8. 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 s</td>
<td>11.35 GB</td>
</tr>
<tr>
<td>New</td>
<td>1.92 s</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. 177)) query. Similarly, you can add a maximum of 100 partitions to a destination table with an INSERT INTO statement. To work around these limitations, 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,
   `l_suppkey` int,
   `l_linenumber` int,
   `l_quantity` double,
   `l_extendedprice` double,
   `l_discount` double,
   `l_tax` double,
   `l_returnflag` string,
   `l_linestatus` 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.

   ```sql
   SHOW PARTITIONS lineitem_parq_partitioned
   ```

   Following are partial sample results.

   ```
   /*
   l_shipdate=1992-01-02
   ```
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.

```sql
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,
       l_linestatus,
       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');
```

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.
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,
    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.

```sql
SHOW PARTITIONS my_lineitem_parq_partitioned;
```

The sample table now has partitions from both January and February 1992.
7. Continue using `INSERT INTO` statements that 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.

---

**Querying with Prepared Statements**

You can use the Athena parameterized query feature to prepare statements for repeated execution of the same query with different query parameters. A prepared statement contains parameter placeholders whose values are supplied at execution time. Prepared statements enable Athena queries to take parameters directly and help to prevent SQL injection attacks.

### Considerations and Limitations

- Prepared statements are workgroup-specific, and prepared statement names must be unique within the workgroup.
- Parameterized queries are supported only in Athena engine version 2. For information about Athena engine versions, see Athena Engine Versioning (p. 459).
- IAM permissions for prepared statements are required. For more information, see Allow Access to Prepared Statements (p. 345).
- Currently, parameterized queries are supported only for `SELECT`, `INSERT INTO`, `CTAS`, and `UNLOAD` statements.

### 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 list of 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><code>statement_name</code></td>
<td>The name of the statement to be prepared. The name must be unique within the workgroup.</td>
</tr>
<tr>
<td><code>statement</code></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

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.
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

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.

Topics

- Summary: Updates and Data Formats in Athena (p. 198)
- Index Access in ORC and Parquet (p. 199)
- Types of Updates (p. 200)
- Updates in Tables with Partitions (p. 205)
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. 199).

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 or TSV, and instead use any of the other supported formats, preferably, a columnar format, such as Parquet or ORC.

### Schema Updates and Data Formats in Athena

<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. 202)</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>N</td>
</tr>
<tr>
<td>Add columns at the beginning or in the middle of the table (p. 201)</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. 202)</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>
<tr>
<td>Remove columns (p. 202)</td>
<td>Store your data in JSON, AVRO, or 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>
</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. 199).
- 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. 200).

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:

```
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'='false')
```

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 Middle of the Table (p. 201)
- Adding Columns at the End of the Table (p. 202)
- Removing Columns (p. 202)
- Renaming Columns (p. 202)
- Reordering Columns (p. 203)
- Changing a Column's Data Type (p. 204)
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:

```
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:

```
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. 199).

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:

```
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  
)  
STORED AS JSON  
LOCATION 's3://schema_updates/orders_json_column_addition/';
```
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. 199).
- 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. 199).
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. 199).

The following example illustrates reordering of columns:
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. 205).

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:

### Compatible Data Types

<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:

CREATE EXTERNAL TABLE orders_json (  'o_orderkey' int,  'o_shippriority' bigint,  ...  )  STORED AS PARQUET  LOCATION 's3://schema_updates/orders_json/';
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. 198).

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. 506) and ALTER TABLE ADD PARTITION (p. 505).

### 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. 206)
- Concatenating Strings and Arrays (p. 208)
- Converting Array Data Types (p. 209)
- Finding Lengths (p. 209)
- Accessing Array Elements (p. 210)
- Flattening Nested Arrays (p. 211)
- Creating Arrays from Subqueries (p. 213)
- Filtering Arrays (p. 214)
- Sorting Arrays (p. 215)
- Using Aggregation Functions with Arrays (p. 215)
- Converting Arrays to Strings (p. 216)
- Using Arrays to Create Maps (p. 217)
- Querying Arrays with Complex Types and Nested Structures (p. 217)

### 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.

```
SELECT ARRAY [1,2,3,4] AS items
```

It returns:

```
+-----------+
<table>
<thead>
<tr>
<th>items</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,2,3,4]</td>
</tr>
</tbody>
</table>
```
Creating Arrays

This query creates two arrays.

```
SELECT ARRAY[ ARRAY[1,2], ARRAY[3,4] ] AS items
```

It returns:

```
+--------------------+
<table>
<thead>
<tr>
<th>items</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[1, 2], [3, 4]]</td>
</tr>
</tbody>
</table>
+--------------------+
```

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:

```
+-----------+
<table>
<thead>
<tr>
<th>items</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,2,3]</td>
</tr>
</tbody>
</table>
+-----------+
```

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:

```
+----------------------------------------+
<table>
<thead>
<tr>
<th>welcome_msg</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[hello, amazon, athena], [hi, alexa]]</td>
</tr>
</tbody>
</table>
+----------------------------------------+
```

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'])
]
```
This query returns:

```
+-----------------------------------------------------------------------------------------------------
| people |
+-----------------------------------------------------------------------------------------------------
+-----------------------------------------------------------------------------------------------------
| [{last=Smith, first=Bob, age=40}, {last=Doe, first=Jane, age=30}, {last=Smith, first=Billy, age=8}] |
+-----------------------------------------------------------------------------------------------------
```

### 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:

```
tables
[[4, 5], [1, 2], [3, 4]]
```

To combine multiple arrays into a single array, use the double pipe operator or the `concat()` function.

WITH
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.

```sql
SELECT
  ARRAY [CAST(4 AS VARCHAR), CAST(5 AS VARCHAR)]
AS items
```

This query returns:

```
+-------+
<table>
<thead>
<tr>
<th>items</th>
</tr>
</thead>
<tbody>
<tr>
<td>[4,5]</td>
</tr>
</tbody>
</table>
+-------+
```

Create two arrays with key-value pair elements, convert them to JSON, and concatenate, as in this example:

```sql
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:

```
+--------------------------------------------------+
<table>
<thead>
<tr>
<th>items</th>
</tr>
</thead>
<tbody>
<tr>
<td>[&quot;a1&quot;:1,&quot;a2&quot;:2,&quot;a3&quot;:3], {&quot;b1&quot;:4,&quot;b2&quot;:5,&quot;b3&quot;:6}]</td>
</tr>
</tbody>
</table>
+--------------------------------------------------+
```

Finding Lengths

The cardinality function returns the length of an array, as in this example:

```sql
SELECT cardinality(ARRAY[1,2,3,4]) AS item_count
```
Amazon Athena User Guide
Accessing Array Elements

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 |
+----------------------------------------+
| hello
| amazon
| athena
|
+----------------------------------------+

210


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:

```
+----------------------+
<table>
<thead>
<tr>
<th>department</th>
<th>names</th>
</tr>
</thead>
<tbody>
<tr>
<td>engineering</td>
<td>Sharon</td>
</tr>
<tr>
<td>-------------</td>
<td>-------</td>
</tr>
<tr>
<td>engineering</td>
<td>John</td>
</tr>
<tr>
<td>-------------</td>
<td>-------</td>
</tr>
<tr>
<td>engineering</td>
<td>Bob</td>
</tr>
<tr>
<td>-------------</td>
<td>-------</td>
</tr>
<tr>
<td>engineering</td>
<td>Sally</td>
</tr>
</tbody>
</table>
+----------------------+
```

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,  
    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
```

211
**Flattening Nested Arrays**

```sql
SELECT names['first'] AS first_name,
       names['last'] AS last_name,
       department FROM dataset
CROSS JOIN UNNEST(people) AS t(names)
```

This query returns:

```
+--------------------------------------+
| first_name | last_name | department |
+--------------------------------------+
| Bob        | Smith     | engineering |
| Jane       | Doe       | engineering |
| Billy      | Smith     | engineering |
+--------------------------------------+
```

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.

```sql
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:

```
+-----------------------------+
| name | department | total_score |
+-----------------------------+
| Amy | devops | 54          |
+-----------------------------+
```

From a list of employees, select the employee with the highest individual score.

```sql
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:

```
+-----------------------------+
| name | department | total_score |
+-----------------------------+
| Amy | devops | 54          |
+-----------------------------+
```

From a list of employees, select the employee with the highest individual score.
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, score FROM users ORDER BY (score) DESC LIMIT 1

This query returns:

+--------------+
| name | score |
+--------------+
| Amy  | 15    |
+--------------+

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
    dataset AS {
        SELECT ARRAY [1,2,2,3,3,4,5] AS items
    }
SELECT array_agg(distinct i) AS array_items
FROM dataset
CROSS JOIN UNNEST(items) AS t(i)

This query returns the following result. Note that ordering is not guaranteed.

+-----------------+
| array_items     |
+-----------------+
| [1, 2, 3, 4, 5] |
+-----------------+
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>[1, 2, 3, 4]</td>
</tr>
</tbody>
</table>

The filter Function

`filter(ARRAY [list_of_values], boolean_function)`

The `filter` function creates an array from the items in the `list_of_values` for which `boolean_function` is true. The `filter` function can be useful in cases in which you cannot use the `UNNEST` function.

The following example creates an array from the values greater than zero in the array `[1,0,5,-1].`
### 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.

```sql
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 on expanding an array into multiple rows, see Flattening Nested Arrays (p. 211).

### 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 Creating Arrays from Subqueries (p. 213).

**Note**

`ORDER BY` is supported for aggregation functions starting in Athena engine version 2.
Converting Arrays to Strings

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.

```
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.

```
+---------------+-----+
| array_items   | total|
+---------------+-----+
| [1, 2, 3, 4]  | 10  |
| [5, 6, 7, 8]  | 26  |
| [9, 0]        | 9   |
+---------------+-----+
```

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`.

```
WITH
  dataset AS {
    SELECT ARRAY ['hello', 'amazon', 'athena'] AS words
  }
SELECT array_join(words, ' ') AS welcome_msg
FROM dataset
```

This query returns:

```
+---------------------+
<table>
<thead>
<tr>
<th>welcome_msg</th>
</tr>
</thead>
</table>
+---------------------+
| 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:

```
+------------+
| first_name |
+------------+
| 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. 218)
• Changing Field Names in Arrays Using CAST (p. 218)
• Filtering Arrays Using the . Notation (p. 219)
• Filtering Arrays with Nested Values (p. 219)
• Filtering Arrays Using UNNEST (p. 220)
• Finding Keywords in Arrays Using regexp_like (p. 221)

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 an AWS CloudTrail logs table by using the dot . notation. For more information, see Querying AWS CloudTrail Logs (p. 276).

```sql
SELECT
    CAST(useridentity.accountid AS bigint) as newid
FROM cloudtrail_logs
LIMIT 2;
```

This query returns:

```
+--------------+
|     newid    |
+--------------+
| 1122345566   |
| 9987655444   |
+--------------+
```

To query an array of values, issue this query:

```sql
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:
Next, to filter and access the BOOLEAN value of that element, continue to use the dot . notation.

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:

<table>
<thead>
<tr>
<th>hostname</th>
<th>isnew</th>
</tr>
</thead>
<tbody>
<tr>
<td>aws.amazon.com</td>
<td>true</td>
</tr>
</tbody>
</table>

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. 211).

For example, this query finds hostnames of sites in the dataset.

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 items  
)  
SELECT sites.hostname, sites.flaggedActivity.isNew  
FROM dataset, UNNEST(items) t(sites)  
WHERE sites.flaggedActivity.isNew = true

It returns:

<table>
<thead>
<tr>
<th>hostname</th>
<th>isnew</th>
</tr>
</thead>
<tbody>
<tr>
<td>aws.amazon.com</td>
<td>true</td>
</tr>
</tbody>
</table>
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 hostname, 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:

```
WITH dataset AS (  
    SELECT ARRAY[  
      CAST(ROW('aws.amazon.com', ROW(ARRAY[  
        MAP(ARRAY['term', 'count'], ARRAY['bigdata', '10']),  
        MAP(ARRAY['term', 'count'], ARRAY['serverless', '50']),  
        MAP(ARRAY['term', 'count'], ARRAY['analytics', '82']),  
        MAP(ARRAY['term', 'count'], ARRAY['iot', '74'])  
      ])) AS ROW(hostname VARCHAR, flaggedActivity ROW(flags ARRAY(MAP(VARCHAR, VARCHAR)) ))  
    ),  
    CAST(  
      ROW('news.cnn.com', ROW(ARRAY[  
        MAP(ARRAY['term', 'count'], ARRAY['politics', '241']),  
        MAP(ARRAY['term', 'count'], ARRAY['technology', '211']),  
        MAP(ARRAY['term', 'count'], ARRAY['serverless', '25']),  
        MAP(ARRAY['term', 'count'], ARRAY['iot', '170'])  
      ])) AS ROW(hostname VARCHAR, flaggedActivity ROW(flags ARRAY(MAP(VARCHAR, VARCHAR)) ))  
    ),  
    CAST(  
      ROW('netflix.com', ROW(ARRAY[  
        MAP(ARRAY['term', 'count'], ARRAY['cartoons', '1020']),  
        MAP(ARRAY['term', 'count'], ARRAY['house of cards', '112042']),  
        MAP(ARRAY['term', 'count'], ARRAY['orange is the new black', '342']),  
        MAP(ARRAY['term', 'count'], ARRAY['iot', '4'])  
      ])) AS ROW(hostname VARCHAR, flaggedActivity ROW(flags ARRAY(MAP(VARCHAR, VARCHAR)) ))  
    ] AS items  
),  
sites AS (  
    SELECT sites.hostname, sites.flaggedactivity  
    FROM dataset, UNNEST(items) t(sites)  
)  
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.

WITH dataset AS (
    SELECT ARRAY[
        CAST(
            ROW('aws.amazon.com', ROW(Array[
                Map(ARRAY['term', 'count'], ARRAY['bigdata', '10']),
                Map(ARRAY['term', 'count'], ARRAY['serverless', '50']),
                Map(ARRAY['term', 'count'], ARRAY['analytics', '82']),
                Map(ARRAY['term', 'count'], ARRAY['iot', '74'])
            ]) AS ROW(hostname VARCHAR, flaggedActivity ROW(flags ARRAY(MAP(VARCHAR, VARCHAR)) ))
        ),
        CAST(
            ROW('news.cnn.com', ROW(Array[
                Map(ARRAY['term', 'count'], ARRAY['politics', '241']),
                Map(ARRAY['term', 'count'], ARRAY['technology', '211']),
                Map(ARRAY['term', 'count'], ARRAY['serverless', '25']),
                Map(ARRAY['term', 'count'], ARRAY['iot', '170'])
            ]) AS ROW(hostname VARCHAR, flaggedActivity ROW(flags ARRAY(MAP(VARCHAR, VARCHAR)) ))
        ),
        CAST(
            ROW('netflix.com', ROW(Array[
                Map(ARRAY['term', 'count'], ARRAY['cartoons', '1020']),
                Map(ARRAY['term', 'count'], ARRAY['house of cards', '112042']),
                Map(ARRAY['term', 'count'], ARRAY['orange is the new black', '342']),
                Map(ARRAY['term', 'count'], ARRAY['iot', '4'])
            ]) AS ROW(hostname VARCHAR, flaggedActivity ROW(flags ARRAY(MAP(VARCHAR, VARCHAR)) ))
        )
    ] AS items
),
sites AS (
    SELECT sites.hostname, sites.flaggedactivity
    FROM dataset, UNNEST(items) t(sites)
)
SELECT hostname, array_agg(flags['term']) AS terms, SUM(CAST(flags['count'] AS INTEGER)) AS total
FROM sites, UNNEST(sites.flaggedActivity.flags) t(flags)
WHERE regexp_like(flags['term'], 'politics|bigdata')
GROUP BY (hostname)
ORDER BY total DESC

This query returns two sites:
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. 223)
• Input Data Formats and Geometry Data Types (p. 223)
• Supported Geospatial Functions (p. 224)
• Examples: Geospatial Queries (p. 246)

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. 146).

**Geometry Data Types**

To handle geospatial queries, Athena supports these specialized geometry data types:

- point
- line
- polygon
- multilinestring
- multipolygon

**Supported Geospatial Functions**

Geospatial functions in Athena have these characteristics:

- The functions follow the general principles of **Spatial Query**.
- The functions are implemented as a Presto plugin that uses the ESRI Java Geometry Library. This library has an Apache 2 license.
- The functions rely on the **ESRI Geometry API**.
- Not all of the ESRI-supported functions are available in Athena. The following topics list only the ESRI geospatial functions that are supported in their respective Athena engine versions. For information about Athena engine versions, see **Athena Engine Versioning** (p. 459).

The geospatial functions that are available in Athena depend on the engine version that you use. For a list of function name changes and new functions in Athena engine version 2, see **Geospatial Function Name Changes and New Functions in Athena engine version 2** (p. 237). For information about Athena engine versioning, see **Athena Engine Versioning** (p. 459).

**Topics**

- Geospatial Functions in Athena engine version 2 (p. 224)
- Geospatial Functions in Athena engine version 1 (p. 239)

**Geospatial Functions in Athena engine version 2**

This topic lists the ESRI geospatial functions that are supported in Athena engine version 2. For a list of AWS Regions that support Athena engine version 2, see **Athena engine version 2** (p. 463). For information about Athena engine versions, see **Athena Engine Versioning** (p. 459).

**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. 469).
The names of some geospatial functions have changed since Athena engine version 1. For more information, see *Geospatial Function Name Changes in Athena engine version 2* (p. 237).

New functions have been added. For more information, see *New Geospatial Functions in Athena engine version 2* (p. 238).

Athena supports the following types of geospatial functions:

- Constructor Functions (p. 225)
- Geospatial Relationship Functions (p. 227)
- Operation Functions (p. 229)
- Accessor Functions (p. 230)
- Aggregation Functions (p. 235)
- Bing Tile Functions (p. 236)

### 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. 224) 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 an Athena engine version 1 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:

```
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 an Athena engine version 1 varbinary type. Example:

```sql
SELECT ST_GeomFromLegacyBinary(ST_GeomAsLegacyBinary(ST_Point(-158.54, 61.56)))
```

**ST_LineFromText(varchar)**

Returns a value in the geometry data type (p. 224) line. Example:

```sql
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:

```sql
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:

```sql
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:

```sql
SELECT ST_Buffer(ST_Point(-74.006801, 40.705220), .072284)
```

Syntax:

```sql
SELECT ST_Point(longitude, latitude) FROM earthquakes LIMIT 1
```

The following example uses specific longitude and latitude coordinates:

```sql
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. 224) polygon. 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:

SELECT ST_Contains('POLYGON((0 2,1 1,0 -1,0 2))', 'POLYGON((-1 3,2 1,0 -3,-1 3))')

SELECT ST_Contains('POLYGON((0 2,1 1,0 -1,0 2))', ST_Point(0, 0))
**ST_Crosses(geography, geography)**

Returns **TRUE** if and only if the left geography crosses the right geography. Example:

```sql
SELECT ST_Crosses(ST_Line('linestring(1 1, 2 2)'), ST_Line('linestring(0 1, 2 2)'))
```

**ST_Disjoint(geography, geography)**

Returns **TRUE** if and only if the intersection of the left geography and the right geography is empty. Example:

```sql
SELECT ST_Disjoint(ST_Line('linestring(0 0, 0 1)'), ST_Line('linestring(1 1, 1 0)'))
```

**ST_Equals(geography, geography)**

Returns **TRUE** if and only if the left geography equals the right geography. Example:

```sql
SELECT ST_Equals(ST_Line('linestring( 0 0, 1 1)'), ST_Line('linestring(1 3, 2 2)'))
```

**ST_Intersects(geography, geography)**

Returns **TRUE** if and only if the left geography intersects the right geography. Example:

```sql
SELECT ST_Intersects(ST_Line('linestring(8 7, 7 8)'), ST_Polygon('polygon((1 1, 4 1, 4 4, 1 4))'))
```

**ST_Overlaps(geography, geography)**

Returns **TRUE** if and only if the left geography overlaps the right geography. Example:

```sql
SELECT ST_Overlaps(ST_Polygon('polygon((2 0, 2 1, 3 1))'), ST_Polygon('polygon((1 1, 1 4, 4 4, 4 1))'))
```

**ST_Relate(geography, geography, varchar)**

Returns **TRUE** if and only if the left geography has the specified dimensionally extended nine-intersection model (DE-9IM) relationship with the right geography. The third (varchar) input takes the relationship. Example:

```sql
SELECT ST_Relate(ST_Line('linestring(0 0, 3 3)'), ST_Line('linestring(1 1, 4 4)'), 'T********')
```

**ST_Touches(geography, geography)**

Returns **TRUE** if and only if the left geography touches the right geography. Example:

```sql
SELECT ST_Touches(ST_Point(8, 8), ST_Polygon('polygon((1 1, 1 4, 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, 1 4, 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)'))
SELECT ST_Boundary(ST_Polygon('polygon((1 1, 1 4, 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 .072284, which is the buffer distance, is specified in angular units as decimal degrees:

```sql
SELECT ST_Buffer(ST_Point(-74.006801, 40.705220), .072284)
```

**ST_Difference(geometry, geometry)**

Returns a geometry of the difference between the left geometry and right geometry. Example:

```sql
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:
**Supported Geospatial Functions**

```sql
SELECT ST_Envelope(ST_Line('linestring(0 1, 1 0)'))

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. 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))

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))

SELECT ST_Intersection(ST_Line('linestring(0 1, 1 0)'), ST_Polygon('polygon((1 1, 1 4, 4 4, 4 1))'))

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_Line('linestring(0 2, 2 2)'), ST_Line('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:

```sql
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, multiline, 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(geometry)**

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))
```

**simplify_geometry(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. 224) polygon, and returns a point geometry data type that is the center of the polygon's envelope. Examples:

```
SELECT ST_Centroid(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. 224), 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. 224), 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. 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. 234). Singular geometries (for example, POINT, LINESTRING, or POLYGON), are treated as collections of one element. Empty geometries are treated as empty collections. Example:

```sql
SELECT ST_GeometryN(ST_Point(-158.54, 61.56),1)
```

**ST_GeometryType(geometry)**

Returns, as a varchar, the type of the geometry. Example:

```sql
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. 234). Example:

```sql
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:

```sql
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. 224). Returns TRUE (type boolean) if and only if the line is closed. Example:

```sql
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. 224). 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:

```sql
SELECT ST_IsEmpty(ST_Point(1.5, 2.5))
```

**ST_IsRing(geometry)**

Returns TRUE (type boolean) if and only if the line type is closed and simple. Example:

```sql
SELECT ST_IsRing(ST_LINE('linestring(0 2, 2 2)'))
```
**ST_IsSimple(geomtry)**

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 `geometry_invalid_reason()` (p. 231). Example:

```sql
SELECT ST_IsSimple(ST_LineString(array[ST_Point(1,2), ST_Point(3,4)]))
```

**ST_IsValid(geomtry)**

Returns true if and only if the specified geometry is well formed. To determine why the geometry is not well formed, use `geometry_invalid_reason()` (p. 231). Example:

```sql
SELECT ST_IsValid(ST_Point(61.56, -86.68))
```

**ST_Length(geomtry)**

Returns the length of line in type double. Example:

```sql
SELECT ST_Length(ST_Line('linestring(0 2, 2 2)'))
```

**ST_NumGeometries(geomtry)**

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:

```sql
ST_NumGeometries(ST_GeometryFromText('GEOMETRYCOLLECTION(MULTIPOINT EMPTY)'))
```

**ST_NumInteriorRing(geomtry)**

Returns the number of interior rings in the polygon geometry in type bigint. Example:

```sql
SELECT ST_NumInteriorRing(ST_Polygon('polygon ((0 0, 8 0, 0 8, 0 0), (1 1, 1 5, 5 1, 1 1))'))
```

**ST_NumPoints(geomtry)**

Returns the number of points in the geometry in type bigint. Example:

```sql
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. 234). Example:

```sql
SELECT ST_PointN(ST_LineString(array[ST_Point(1,2), ST_Point(3,4)]),1)
```

**ST_Points(geomtry)**

Returns an array of points from the specified line string geometry object. Example:
Supported Geospatial Functions

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:

```
SELECT bing_tile(10, 20, 12)
```

**`bing_tile(quadKey)`**

Returns a Bing tile object from a quadkey. Example:

```
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:

```
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:

```
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:

```
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 cover 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 a list of AWS Regions that support Athena engine version 2, see Athena engine version 2 (p. 463).

For information about other changes in Athena engine version 2, see Athena engine version 2 (p. 463).

For information about Athena engine versioning, see Athena Engine Versioning (p. 459).

#### 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>Athena engine version 1 Function Name</th>
<th>Athena engine version 2 Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>st_coordinate_dimension (p. 244)</td>
<td>ST_CoordDim (p. 232)</td>
</tr>
<tr>
<td>st_end_point (p. 244)</td>
<td>ST_EndPoint (p. 232)</td>
</tr>
<tr>
<td>st_exterior_ring (p. 243)</td>
<td>ST_ExtteriorRing (p. 230)</td>
</tr>
<tr>
<td>st_interior_ring_number (p. 245)</td>
<td>ST_NumInteriorRing (p. 234)</td>
</tr>
<tr>
<td>st_geometry_from_text (p. 239)</td>
<td>ST_GeometryFromText (p. 225)</td>
</tr>
<tr>
<td>st_is_closed (p. 245)</td>
<td>ST_IsClosed (p. 233)</td>
</tr>
<tr>
<td>st_is_empty (p. 245)</td>
<td>ST_IsEmpty (p. 233)</td>
</tr>
<tr>
<td>st_is_ring (p. 245)</td>
<td>ST_IsRing (p. 233)</td>
</tr>
<tr>
<td>st_max_x (p. 245)</td>
<td>ST_XMax (p. 235)</td>
</tr>
<tr>
<td>st_max_y (p. 245)</td>
<td>ST_YMax (p. 235)</td>
</tr>
<tr>
<td>st_min_x (p. 245)</td>
<td>ST_XMin (p. 235)</td>
</tr>
<tr>
<td>st_min_y (p. 246)</td>
<td>ST_YMin (p. 235)</td>
</tr>
</tbody>
</table>
Supported Geospatial Functions

<table>
<thead>
<tr>
<th>Athena engine version 1 Function Name</th>
<th>Athena engine version 2 Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>st_point_number (p. 246)</td>
<td>ST_NumPoints (p. 234)</td>
</tr>
<tr>
<td>st_start_point (p. 246)</td>
<td>ST_StartPoint (p. 235)</td>
</tr>
<tr>
<td>st_symmetric_difference (p. 244)</td>
<td>ST_SymDifference (p. 230)</td>
</tr>
</tbody>
</table>

New Geospatial Functions in Athena engine version 2

The following geospatial functions are new in Athena engine version 2. For more information, visit the corresponding links.

**Constructor Functions**
- ST_AsBinary (p. 225)
- ST_GeomAsLegacyBinary (p. 225)
- ST_GeomFromBinary (p. 225)
- ST_GeomFromLegacyBinary (p. 226)
- ST_LineString (p. 226)
- ST_MultiPoint (p. 226)
- to_geometry (p. 227)
- to_spherical_geography (p. 227)

**Operation Functions**
- geometry_union (p. 229)
- ST_EnvelopeAsPts (p. 230)
- ST_Union (p. 230)

**Accessor Functions**
- geometry_invalid_reason (p. 231)
- great_circle_distance (p. 231)
- line_locate_point (p. 231)
- simplify_geometry (p. 231)
- ST_ConvexHull (p. 231)
- ST_Distance (spherical geography) (p. 232)
- ST_Geometries (p. 232)
- ST_GeometryN (p. 233)
- ST_GeometryType (p. 233)
- ST_InteriorRingN (p. 233)
- ST_InteriorRings (p. 233)
- ST_IsSimple (p. 234)
- ST_IsValid (p. 234)
- ST_NumGeometries (p. 234)
- ST_PointN (p. 234)
- ST_Points (p. 234)
Aggregation Functions

- convex_hull_agg (p. 235)
- geometry_union_agg (p. 235)

Bing Tile Functions

- bing_tile (p. 236)
- bing_tile (quadkey) (p. 236)
- bing_tile_at (p. 236)
- bing_tiles_around (p. 236)
- bing_tiles_around (radius) (p. 236)
- bing_tile_coordinates (p. 236)
- bing_tile_polygon (p. 236)
- bing_tile_quadkey (p. 236)
- bing_tile_zoom_level (p. 237)
- geometry_to_bing_tiles (p. 237)

Geospatial Functions in Athena engine version 1

Geospatial functions in Athena have the following characteristics:

- The functions follow the general principles of Spatial Query.
- The functions are implemented as a Presto plugin that uses the ESRI Java Geometry Library. This library has an Apache 2 license.
- The functions rely on the ESRI Geometry API.
- Not all of the ESRI-supported functions are available in Athena. This topic lists only the ESRI geospatial functions that are supported in Athena engine version 1.

Athena supports four types of geospatial functions:

- Constructor Functions (p. 239)
- Geospatial Relationship Functions (p. 241)
- Operation Functions (p. 242)
- Accessor Functions (p. 244)

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_GEOMETRY_FROM_TEXT (varchar)**

Converts text into a geometry data type. Returns a value in a varbinary data type, which is a binary representation of the geometry data type. Example:

```
SELECT ST_GEOMETRY_FROM_TEXT(ST_GEOMETRY_TO_TEXT(ST_Point(1, 2)))
```
**ST_GEOMETRY_TO_TEXT** *(varbinary)*

Converts each of the specified geometry data types (p. 224) to text. Returns a value in a varchar data type, which is a WKT representation of the geometry data type. Example:

```sql
SELECT ST_GEOMETRY_TO_TEXT(ST_Point(-158.54, 61.56))
```

**ST_LINE**(varchar)

Returns a value in the varbinary data type, which is a binary representation of the geometry data type (p. 224) line. Example:

```sql
SELECT ST_Line('linestring(1 1, 2 2, 3 3)')
```

**ST_POINT**(double, double)

Returns a value in the varbinary data type, which is a binary representation of a point geometry data type.

To obtain the point geometry data type, use the **ST_POINT** function in Athena. 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:

```sql
ST_BUFFER(ST_POINT(-74.006801, 40.705220), .072284)
```

Syntax:

```sql
SELECT ST_POINT(longitude, latitude) FROM earthquakes LIMIT 1;
```

Example. This example uses specific longitude and latitude coordinates from earthquakes.csv:

```sql
SELECT ST_Point(-158.54, 61.56) FROM earthquakes LIMIT 1;
```

It returns this binary representation of a geometry data type point:

```
00 00 00 00 01 01 00 00 00 48 e1 7a 14 ae c7 4e 40 e1 7a 14 ae 47 d1 63 c0
```

The next example uses specific longitude and latitude coordinates:

```sql
SELECT ST_POINT(-74.006801, 40.705220);
```

It returns this binary representation of a geometry data type point:

```
00 00 00 00 01 01 00 00 00 20 25 76 6d 6f 80 52 c0 18 3e 22 a6 44 5a 44 40
```

In the following example, we use the **ST_GEOMETRY_TO_TEXT** function to obtain the binary values from WKT:
Supported Geospatial Functions

**ST_POLYGON(varchar)**

Using the sequence of the ordinates provided clockwise, left to right, returns a value in the varbinary data type, which is a binary representation of the `geometry` data type (p. 224) polygon. Example:

```sql
SELECT ST_POLYGON('polygon ((1 1, 1 4, 4 4, 4 1))')
```

**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. In Athena engine version 1, geospatial relationship function inputs are representations of geometries in varchar or varbinary format.

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:

```sql
SELECT ST_CONTAINS('POLYGON((0 2,1 1,0 -1,0 2))', 'POLYGON((-1 3,2 1,0 -3,-1 3))')
SELECT ST_CONTAINS('POLYGON((0 2,1 1,0 -1,0 2))', ST_Point(0, 0));
SELECT ST_CONTAINS(ST_GEOMETRY_FROM_TEXT('POLYGON((0 2,1 1,0 -1,0 2))'),
                  ST_GEOMETRY_FROM_TEXT('POLYGON((-1 3,2 1,0 -3,-1 3))'))
```

**ST_CROSSES (geometry, geometry)**

Returns **TRUE** if and only if the left geometry crosses the right geometry. Example:

```sql
SELECT ST_CROSSES(ST_LINE('linestring(1 1, 2 2)'), ST_LINE('linestring(0 1, 2 2)'))
```

**ST_DISJOINT (geometry, geometry)**

Returns **TRUE** if and only if the intersection of the left geometry and the right geometry is empty. Example:

```sql
SELECT ST_DISJOINT(ST_LINE('linestring(0 0, 0 1)'), ST_LINE('linestring(1 1, 1 0)'))
```

**ST_EQUALS (geometry, geometry)**

Returns **TRUE** if and only if the left geometry equals the right geometry. Example:

```sql
SELECT ST_EQUALS(ST_LINE('linestring( 0 0, 1 1)'), ST_LINE('linestring(1 3, 2 2)'))
```
**ST_INTERSECTS** *(geometry, geometry)*

Returns **TRUE** if and only if the left geometry intersects the right geometry. Example:

```sql
SELECT ST_INTERSECTS(ST_LINE('linestring(8 7, 7 8)'), ST_POLYGON('polygon((1 1, 4 4, 1 4))'))
```

**ST_OVERLAPS** *(geometry, geometry)*

Returns **TRUE** if and only if the left geometry overlaps the right geometry. Example:

```sql
SELECT ST_OVERLAPS(ST_POLYGON('polygon((2 0, 2 1, 3 1))'), ST_POLYGON('polygon((1 1, 1 4, 4 4, 4 1))'))
```

**ST_RELATE** *(geometry, geometry)*

Returns **TRUE** if and only if the left geometry has the specified Dimensionally Extended nine-Intersection Model (DE-9IM) relationship with the right geometry. For more information, see the Wikipedia topic [DE-9IM]. Example:

```sql
SELECT ST_RELATE(ST_LINE('linestring(0 0, 3 3)'), ST_LINE('linestring(1 1, 4 4)'), 'T********')
```

**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, 1 4, 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, 1 4, 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.

In Athena engine version 1, all operation functions take one of the geometry data types as an input and return a binary representation as a varbinary data type.

**ST_BOUNDARY** *(geometry)*

Takes as an input one of the geometry data types, and returns a binary representation of the boundary geometry data type. Examples:

```sql
SELECT ST_BOUNDARY(ST_LINE('linestring(0 1, 1 0)'))
```
SELECT ST_BOUNDARY(ST_POLYGON('polygon((1 1, 1 4, 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 a binary representation of the geometry data type buffered by the specified distance (or radius). Example:

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 `.072284`, which is the buffer distance, is specified in angular units as decimal degrees:

ST_BUFFER(ST_POINT(-74.006801, 40.705220), .072284)

**ST_DIFFERENCE (geometry, geometry)**

Returns a binary representation of a difference between the left geometry and right geometry. Example:

SELECT ST_GEOMETRY_TO_TEXT(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 a binary representation of an envelope, where an envelope is a rectangle around the specified geometry data type. Examples:

SELECT ST_ENVELOPE(ST_LINE('linestring(0 1, 1 0)'))

SELECT ST_ENVELOPE(ST_POLYGON('polygon((1 1, 4 4, 4 1))'))

**ST_EXTERIOR_RING (geometry)**

Returns a binary representation of the exterior ring of the input type `polygon`. Examples:

SELECT ST_EXTERIOR_RING(ST_POLYGON(1,1, 1,4, 4,1))

SELECT ST_EXTERIOR_RING(ST_POLYGON('polygon ((0 0, 8 0, 0 8, 0 0), (1 1, 5 1, 1 1))'))

**ST_INTERSECTION (geometry, geometry)**

Returns a binary representation of the intersection of the left geometry and right geometry. Examples:

SELECT ST_INTERSECTION(ST_POINT(1,1), ST_POINT(1,1))

SELECT ST_INTERSECTION(ST_LINE('linestring(0 1, 1 0)'), ST_POLYGON('polygon((1 1, 1 4, 4 4, 4 1))'))

SELECT ST_GEOMETRY_TO_TEXT(ST_INTERSECTION(ST_POLYGON('polygon((2 0, 2 3, 3 0))'), ST_POLYGON('polygon((1 1, 4 1, 4 4, 1 4))')))

243
**ST_SYMMETRIC_DIFFERENCE** *(geometry, geometry)*

Returns a binary representation of the geometrically symmetric difference between left geometry and right geometry. Example:

```sql
SELECT ST_GEOMETRY_TO_TEXT(ST_SYMMETRIC_DIFFERENCE(ST_LINE('linestring(0 2, 2 2)'), ST_LINE('linestring(1 2, 3 2)')))
```

**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.

**ST_AREA** *(geometry)*

Takes as an input a geometry data type `polygon` and returns an area in type `double`. Example:

```sql
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. 224) `polygon`, and returns a point that is the center of the polygon's envelope in type `varchar`. Examples:

```sql
SELECT ST_CENTROID(ST_GEOMETRY_FROM_TEXT('polygon ((0 0, 3 6, 6 0, 0 0))'))
```

```sql
SELECT ST_GEOMETRY_TO_TEXT(ST_CENTROID(ST_ENVELOPE(ST_GEOMETRY_FROM_TEXT('POINT (53 27)'))))
```

**ST_COORDINATE_DIMENSION** *(geometry)*

Takes as input one of the supported geometry data types (p. 224), and returns the count of coordinate components in type `bigint`. Example:

```sql
SELECT ST_COORDINATE_DIMENSION(ST_POINT(1.5,2.5))
```

**ST_DIMENSION** *(geometry)*

Takes as an input one of the supported geometry data types (p. 224), and returns the spatial dimension of a geometry in type `bigint`. Example:

```sql
SELECT ST_DIMENSION(ST_POLYGON('polygon((1 1, 4 1, 4 4, 1 4))'))
```

**ST_DISTANCE** *(geometry, geometry)*

Returns, based on spatial ref, the two-dimensional minimum Cartesian distance between two geometries in projected units. Example:

```sql
SELECT ST_DISTANCE(ST_POINT(0.0,0.0), ST_POINT(3.0,4.0))
```

**ST_END_POINT** *(geometry)*

Returns the last point of a line geometry data type in type `point`. Example:
**Supported Geospatial Functions**

```sql
SELECT ST_END_POINT(ST_LINE('linestring(0 2, 2 2)'))
```

**ST_INTERIOR_RING_NUMBER (geometry)**

Returns the number of interior rings in the polygon geometry in type bigint. Example:

```sql
SELECT ST_INTERIOR_RING_NUMBER(ST_POLYGON('polygon ((0 0, 8 0, 0 8, 0 0), (1 1, 1 5, 5 1, 1 1))'))
```

**ST_IS_CLOSED (geometry)**

Takes as an input only line and multiline geometry data types (p. 224). Returns TRUE (type boolean) if and only if the line is closed. Example:

```sql
SELECT ST_IS_CLOSED(ST_LINE('linestring(0 2, 2 2)'))
```

**ST_IS_EMPTY (geometry)**

Takes as an input only line and multiline geometry data types (p. 224). 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:

```sql
SELECT ST_IS_EMPTY(ST_POINT(1.5, 2.5))
```

**ST_IS_RING (geometry)**

Returns TRUE (type boolean) if and only if the line type is closed and simple. Example:

```sql
SELECT ST_IS_RING(ST_LINE('linestring(0 2, 2 2)'))
```

**ST_LENGTH (geometry)**

Returns the length of line in type double. Example:

```sql
SELECT ST_LENGTH(ST_LINE('linestring(0 2, 2 2)'))
```

**ST_MAX_X (geometry)**

Returns the maximum X coordinate of a geometry in type double. Example:

```sql
SELECT ST_MAX_X(ST_LINE('linestring(0 2, 2 2)'))
```

**ST_MAX_Y (geometry)**

Returns the maximum Y coordinate of a geometry in type double. Example:

```sql
SELECT ST_MAX_Y(ST_LINE('linestring(0 2, 2 2)'))
```

**ST_MIN_X (geometry)**

Returns the minimum X coordinate of a geometry in type double. Example:

```sql
SELECT ST_MIN_X(ST_LINE('linestring(0 2, 2 2)'))
```
ST_MIN_Y (geometry)
Returns the minimum Y coordinate of a geometry in type double. Example:

```
SELECT ST_MAX_Y(ST_LINE('linestring(0 2 2 2)'))
```

ST_POINT_NUMBER (geometry)
Returns the number of points in the geometry in type bigint. Example:

```
SELECT ST_POINT_NUMBER(ST_POINT(1.5, 2.5))
```

ST_START_POINT (geometry)
Returns the first point of a line geometry data type in type point. Example:

```
SELECT ST_START_POINT(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_Y (point)
Returns the Y coordinate of a point in type double. Example:

```
SELECT ST_Y(ST_POINT(1.5, 2.5))
```

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:

```
CREATE external TABLE earthquakes
(
  earthquake_date string,
  latitude double,
  longitude double,
  depth double,
  magnitude double,
  magtype string,
```
The following code example creates a table called `counties`:

```sql
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. The query uses Athena engine version 2.

**Note**

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. 226) function to convert the `boundaryshape` binary value into a geometry. If you are using Athena engine version 1, you do not need to do this conversion. For more information, see Changes to Geospatial Functions (p. 469) in the Athena engine version 2 (p. 463) 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 |
+------------------------+---+
```
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:
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 Types & Queries 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 on 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. 177) or INSERT INTO (p. 479) 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 Hudi Tables 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. 505).

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.

```
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,
  `event_name` string,
) WITH ( tid = "1", iar = true, coar = true, early_coar = true, output_format = "Parquet",
  s3_output_path = "s3://happybucket/happytable/chicagofield/" );
```
Partitioned CoW Table

The following example creates a partitioned CoW table in Athena.

```sql
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.

```sql
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`.

```sql
CREATE EXTERNAL TABLE `nonpartition_mor`(
  `_hoodie_commit_time` string,
  `_hoodie_commit_seqno` string,
```
The following example creates a nonpartitioned MoR table in Athena for snapshot queries. For snapshot queries, use the input format `HoodieParquetRealtimeInputFormat`.

```sql
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.realtime.HoodieParquetRealtimeInputFormat'
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.

```sql
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,
    `event_type` string)
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/nonpartition_mor'
```
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`(
  `_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.realtime.HoodieParquetRealtimeInputFormat'
OUTPUTFORMAT
  'org.apache.hudi.hadoop.hive.gl.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 on 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. 254)
- Extracting Data from JSON (p. 255)
- Searching for Values in JSON Arrays (p. 257)
- Obtaining Length and Size of JSON Arrays (p. 259)
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. 146).

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. 146).
- Make sure that each JSON-encoded record is represented on a separate line.
- 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. 146).

  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. 254)
- Converting JSON to Athena Data Types (p. 255)

Converting Athena Data Types to JSON

To convert Athena data types to JSON, use `CAST`.

```sql
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
```

254
Extracting Data from JSON

This query returns:

```
+-------------------------------------------+
| hello_msg      | some_int | some_map      |
+-------------------------------------------+
| "HELLO ATHENA" | 12345    | {"a":1,"b":2} |
+-------------------------------------------+
```

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:

```
+-------------------------------------------+
| hello_msg      | some_int | some_map      |
+-------------------------------------------+
| HELLO ATHENA   | 12345    | {a:1,b:2}     |
+-------------------------------------------+
```

Extracting Data from JSON

You may have source data with 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.

```
{"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.

<table>
<thead>
<tr>
<th>name</th>
<th>projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Susan Smith&quot;</td>
<td>[&quot;project1&quot;, &quot;project2&quot;]</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>name</th>
<th>projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Susan Smith</td>
<td></td>
</tr>
</tbody>
</table>

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": "marketing",
    "projects": [{"name": "project1", "completed":false},
    {"name": "project2", "completed":true}]}
    AS blob
)
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": "engineering", "projects": [{"name": "project1", "completed": false}, {"name": "project2", "completed": true}]'}),
        (JSON '{"name": "Susan Smith", "org": "engineering", "projects": [{"name": "project1", "completed": false}, {"name": "project2", "completed": true}]'}),
        (JSON '{"name": "Jane Smith", "org": "finance", "projects": [{"project1", "project2"]}') AS t (users)
    )
    SELECT json_extract_scalar(users, '$.name') AS user
    FROM dataset

    WHERE json_array_contains(json_extract(users, '$.projects'), 'project2')
```

This query returns a list of users.
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                 |
| Jane Smith  | 1                 |
+------------------------+------------------+
```
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, '$.name') as name,
    json_size(users, '$.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. 532) 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. 528).

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 available in the Regions where Athena engine version 2 is supported. For a list of AWS Regions that support Athena engine version 2, see Athena engine version 2 (p. 463).
- **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 in the Amazon SageMaker Developer Guide and CreateEndpointConfig 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 (p. 362).
- 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)
RETURNS DOUBLE
```
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.

Detect Botnets

The following video shows how one company uses Amazon Athena and Amazon SageMaker to detect botnets.

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. 265). 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 Regions where Athena engine version 2 is supported. For a list of AWS Regions that support Athena engine version 2, see Athena engine version 2 (p. 463).
• **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 [Presto Functions in Amazon Athena](#) (p. 492).

• **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.

• **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. 358).

• **Lambda quotas** – Lambda quotas apply to UDFs. For more information, see [Lambda quotas](#) in the AWS Lambda Developer Guide.

• **Known issues** – For the most up-to-date list of known issues, see [Limitations and Issues](#) in the awslabs/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**

```
USING EXTERNAL FUNCTION UDF_name(variable1 data_type[, variable2 data_type][,...])
RETURNS data_type
```
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.

<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 (see RETURNS note)</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>

**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. 265)
• Create your Maven Project (p. 266)
• Add Dependencies and Plugins to Your Maven Project (p. 266)
• Write Java Code for the UDFs (p. 267)
• Build the JAR File (p. 268)
• Deploy the JAR to AWS Lambda (p. 268)

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. 68).

  ```
  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 \
    -DgroupId=groupId \
    -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.

```
<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.
     * @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.
     * @param input the String that has been compressed
     * @return the decompressed String
     */
    public String decompress(String input)
    {
        byte[] compressedBytes = Base64.getDecoder().decode(input);

        // create decompressor
        Inflater decompressor = new Inflater();
        decompressor.setInput(compressedBytes);

        // decompress bytes to output stream
        byte[] buffer = new byte[4096];
        ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream(compressedBytes.length);
        while (!decompressor.finished()) {
            int bytes = decompressor.deflate(buffer);
            byteArrayOutputStream.write(buffer, 0, bytes);
        }

        try {
            byteArrayOutputStream.close();
        } catch (IOException e) {
            throw new RuntimeException("Failed to close ByteArrayOutputStream", e);
        }

        // return decompressed string
        return new String(byteArrayOutputStream.toByteArray());
    }
}
```
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 athena-query-federation/tools directory of the Athena Query Federation SDK that you cloned earlier.

For more information and requirements, see Publishing Applications in the AWS Serverless Application Repository Developer Guide, AWS SAM Template Concepts in the AWS Serverless Application Model Developer Guide, and Publishing Serverless Applications Using the AWS SAM CLI.

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

269
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::1234567890123:role/my_lambda_role \
--handler com.mycompany.athena.udfs.MyUserDefinedFunctions \
--timeout 900 \
--zip-file fileb://./target/my-athena-udfs-1.0-SNAPSHOT.jar
```

**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. 87), 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 on 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. 6) and Getting Started (p. 8).

**Topics**

- Querying Application Load Balancer Logs (p. 271)
- Querying Classic Load Balancer Logs (p. 273)
- Querying Amazon CloudFront Logs (p. 274)
- Querying AWS CloudTrail Logs (p. 276)
- Querying Amazon EMR Logs (p. 282)
- Querying AWS Global Accelerator Flow Logs (p. 285)
- Querying Amazon GuardDuty Findings (p. 287)
- Querying AWS Network Firewall Logs (p. 289)
- Querying Network Load Balancer Logs (p. 290)
- Querying Amazon Route 53 Resolver Query Logs (p. 291)
- Querying Amazon VPC Flow Logs (p. 294)
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. 271)
• Creating the Table for ALB Logs (p. 271)
• Example Queries for ALB Logs (p. 272)

Prerequisites

• Enable access logging so that Application Load Balancer logs can be saved to your Amazon S3 bucket.

Creating the Table for ALB Logs

1. Copy and paste the following CREATE TABLE statement into the Athena console. Replace the values in LOCATION 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.

```
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 string,  
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,
```

271
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.

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 example shows how to parse the logs by datetime:

```sql
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;
```

- For more information and examples, see the AWS Knowledge Center article How do I analyze my Application Load Balancer access logs using Athena?
- For more information about partitioning ALB logs with Athena, see athena-add-partition on GitHub.
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. 273)
- Elastic Load Balancing Example Queries (p. 273)

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.

   ```sql
   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-Z0-9-]+) (\[A-Za-z0-9.-]+)\$'
   )
   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. 273)

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. 274)
• Example Query for CloudFront logs (p. 275)

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. 8).

This query uses the default SerDe, LazySimpleSerDe (p. 150). The column date is escaped using backticks (`) because it is a reserved word in Athena. For information, see Reserved Keywords (p. 107).

```
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_start BIGINT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LOCATION 's3://CloudFront_bucket_name/CloudFront/'
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.

```
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.

```
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.
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.

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. 276)
- Using the CloudTrail Console to Create an Athena Table for CloudTrail Logs (p. 277)
- Creating the Table for CloudTrail Logs in Athena Using Manual Partitioning (p. 278)
- Creating the Table for CloudTrail Logs in Athena Using Partition Projection (p. 279)
- Querying Nested Fields (p. 280)
- Example Query (p. 281)
- Tips for Querying CloudTrail Logs (p. 281)

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/'.

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. 6).
- For information about creating a table with partitions, see Creating the 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 the 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. The statement is the same as the one in the CloudTrail console **Create a table in Amazon Athena** dialog box, but adds a **PARTITIONED** BY clause that makes the table partitioned.

2. Modify `s3://CloudTrail_bucket_name/AWSLogs/Account_ID/CloudTrail/` to point to the Amazon S3 bucket that contains your log data.

3. Verify that fields are listed correctly. For more information about the full list of fields in a CloudTrail record, see **CloudTrail Record Contents**.

   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. 255). For performance improvements, this example partitions the data by Region, year, month, and day.

   ```sql
   CREATE EXTERNAL TABLE cloudtrail_logs (  
     eventversion STRING,  
     useridentity STRUCT<  
       type:STRING,  
       principalid:STRING,  
       arn:STRING,  
       accountid:STRING,  
       invokedby:STRING,  
       accesseskeyid:STRING,  
       userName:STRING,  
     sessioncontext:STRUCT<  
       attributes:STRUCT<  
         mfaauthenticated:STRING,  
         creationdate:STRING>,  
     sessionissuer:STRUCT<  
       type:STRING,  
       principalid:STRING,  
       arn:STRING,  
       accountid:STRING,  
       userName: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,  
   )
   ```

278
### Amazon Athena User Guide

#### CloudTrail

```sql
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,
        ...
      >
    >
  >
)
PARTITIONED BY (region string, year string, month string, day string)
ROW FORMAT SERDE 'com.amazon.emr.hive.serde.CloudTrailSerde'
STORED AS INPUTFORMAT 'com.amazon.emr.cloudtrail.CloudTrailInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION 's3://CloudTrail_bucket_name/AWSLogs/Account_ID/CloudTrail/';
```

4. Run the query in the Athena console.

5. Use the `ALTER TABLE ADD PARTITION` (p. 505) command to load the partitions so that you can query them, as in the following example.

```sql
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 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.

```sql
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,
        ...
      >
    >
  >
)
```
For more information about partition projection, see Partition Projection with Amazon Athena (p. 119).

**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
    eventdatacreatetime,
    eventdataeventname,
    eventdatauserIdentity.sessioncontext.attributes.creationdate,
    eventdatauserIdentity.sessioncontext.sessionissuer.arn
```
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.

```sql
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.

```sql
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. 214).

### 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:

```sql
SELECT *
FROM cloudtrail_logs
WHERE
    eventsource = 's3.amazonaws.com' AND
    eventname in ('GetObject') AND
    useridentity.accountid LIKE '%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 the Table for CloudTrail Logs in Athena Using Manual Partitioning” (p. 278). 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 the Table for CloudTrail Logs (p. 278).

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. 255).

Now you are ready to issue 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.

```
SELECT
    useridentity.arn,
    eventname,
    sourceipaddress,
    eventtime
FROM cloudtrail_logs
LIMIT 100;
```

- Modify the earlier 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. 283).

```
CREATE EXTERNAL TABLE `myemrlogs`(
    `data` string COMMENT 'from deserializer')
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|' LINES TERMINATED BY '
' 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-00b3c0a839ee0a9c, for ERROR, WARN, INFO, EXCEPTION, FATAL, or DEBUG

```sql
SELECT "data",
"$PATH" AS filepath
FROM "default"."myemrlogs"
WHERE regexp_like("$PATH",'i-00b3c0a839ee0a9c')
    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. 114).
The following query creates the partitioned table named `mypartitionedemrlogs`:

```sql
CREATE EXTERNAL TABLE `mypartitionedemrlogs`(
  `data` string COMMENT 'from deserializer')
partitioned by (logtype string)
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 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')

ALTER TABLE mypartitionedemrlogs ADD
  PARTITION (logtype='hadoop-mapreduce')

ALTER TABLE mypartitionedemrlogs ADD
  PARTITION (logtype='hadoop-state-pusher')

ALTER TABLE mypartitionedemrlogs ADD
  PARTITION (logtype='node')
  LOCATION 's3://aws-logs-123456789012-us-west-2/elasticmapreduce/j-2ABCDE34F5GH6/node/'

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') limit 100;
```
Example – Querying the Hadoop-Mapreduce Partition for Job job_1561661818238_0004 and Failed Reduces

```sql
SELECT data,
"$PATH"
FROM "default"."mypartitionedemrlogs"
WHERE logtype='hadoop-mapreduce'
    AND regexp_like(data,'job_1561661818238_0004|Failed Reduces') limit 100;
```

Example – Querying Hive Logs in the Node Partition for Query ID 056e0609-33e1-4611-956c-7a31b42d2663

```sql
SELECT data,
"$PATH"
FROM "default"."mypartitionedemrlogs"
WHERE logtype='node'
    AND regexp_like("$PATH','hive')
    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

```sql
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. 133), which means that the query uses the `LazySimpleSerDe` (p. 150). In this query, fields are terminated by a space.

```sql
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,
    ...)
```

285
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/'
```

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.

```
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.

```
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.
SELECT SUM(numpackets) AS packetcount, endpointip 
FROM aga_flow_logs 
WHERE endpointport = 443 
GROUP BY endpointip 
ORDER BY packetcount DESC 
LIMIT 10;

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,  `createdate` string,  `updatedate` 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')
```
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,
    createdate,
    updatedate,
    json_extract_scalar(service, '$.count') AS Count,
    json_extract_scalar(resource, '$.instancetype.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,
    createdate,
    updatedate,
    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. 255).
- Make sure that all characters in the JSON fields are in lower case.
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. 107) 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:int,
         start:string,
         finish:string,
         age:int,
         min_ttl:int,
         max_ttl:int
       >
     >
   )
   ROW FORMAT SERDE
   'org.openx.data.jsonserde.JsonSerDe'
   WITH SERDEPROPERTIES (
     'paths'='availability_zone,event,event_timestamp,firewall_name'
   )
   STORED AS INPUTFORMAT
   'org.apache.hadoop.mapred.TextInputFormat'
   ```
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. 290)
- Network Load Balancer Example Queries (p. 291)

### 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,
    ...)
```
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 the older TLS version:

```sql
SELECT tls_protocol_version, COUNT(tls_protocol_version) AS num_connections, client_ip FROM "nlb_tls_logs" WHERE tls_protocol_version < 'tlsv12' 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;
```

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 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 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 (p. 146) to analyze the data.

   **Note**
   The SerDe expects each JSON record in the logs in Amazon S3 to be on a single line of text with no line termination characters separating the fields in the record. If the log JSON text is in pretty print format, you may receive the error message HIVE_CURSOR_ERROR: Row is not a valid JSON Object when you attempt to query the table after you create it.

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.
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" LIMIT 10
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('2020-09-24-00:08:00','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.

```
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 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.

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 you can specify a custom format. A custom format is where you specify which fields to return in the flow log, and the order in which they should appear. For more information, see Flow Log Records in the Amazon VPC User Guide.

- Creating the Table for VPC Flow Logs (p. 294)
- Example Queries for Amazon VPC Flow Logs (p. 295)

Creating the Table for VPC Flow Logs

The following procedure creates an Amazon VPC table for 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 the Amazon VPC table

1. Copy and paste a DDL statement like the following into the Athena console Query Editor. The following sample statement creates a table that has the columns for 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 this sample CREATE TABLE statement accordingly.

```sql
CREATE EXTERNAL TABLE IF NOT EXISTS vpc_flow_logs (  
  version int,
  account string,
  interfaceid string,
  sourceaddress string,
  destinationaddress string,
  sourceport int,
  destinationport int,
  protocol int,
  numpackets int,
  numbytes bigint,
  starttime int,
  endtime int,
  action string,
  logstatus string,
  vpcid string,
  subnetid string,
  instanceid string,
  tcpflags int,
  type string,
  pktsrcaddr string,
  pktdstaddr string,
  region string,
  azid string,
  sublocationtype string,
  sublocationid string,
  pktsrcawsservice string,
  pktdstawsservice string,
  flowdirection string,
) WITH (  
  format='GZIP',
  location='s3://bucketname/flowlogs/
);```
trafficpath string
)
PARTITIONED BY (`date` date)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ' ' 
LOCATION 's3://your_log_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. 150). 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. 107).

- 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://your_log_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.

   ```sql
   ALTER TABLE vpc_flow_logs
   ADD PARTITION (`date`='YYYY-MM-dd')
   location 's3://your_log_bucket/prefix/AWSLogs/{account_id}/vpcflowlogs/{region_code}/YYYY/MM/dd';
   ```

**Example Queries for Amazon VPC Flow Logs**

The following example query lists a maximum of 100 flow logs for the date specified.

```sql
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.

```sql
SELECT day_of_week(date) AS day,
   date,
   interfaceid,
   sourceaddress,
   action,
   protocol
```
To see which one of your servers is receiving the highest number of HTTPS requests, use this 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(numpackets) AS packetcount, destinationaddress
FROM vpc_flow_logs
WHERE destinationport = 443 AND date > current_date - interval '7' day
GROUP BY destinationaddress
ORDER BY packetcount DESC
LIMIT 10;
```

For more information, see the AWS Big Data blog post Analyzing VPC Flow Logs with Amazon Kinesis Firehose, Athena, and Amazon QuickSight. The blog post uses version 2 of the VPC flow logs.

### 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 web ACL traffic information 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 Amazon ES, Amazon Athena, and Amazon QuickSight.

### Topics

- Creating the Table for AWS WAF Logs (p. 296)
- Creating the Table for AWS WAF Logs in Athena Using Partition Projection (p. 298)
- Example Queries for AWS WAF Logs (p. 300)

### Creating the Table for AWS WAF Logs

#### To create the AWS WAF table

1. Copy and paste the following DDL statement into the Athena console. Modify the `LOCATION` for the Amazon S3 bucket that stores your logs.

   ```sql
   CREATE TABLE waf_table (
     logentryid string,
     ruleName string,
     requestTime string,
     bytesFromSource string,
     bytesToSource string,
     httpMethod string,
     httpReason string,
     httpCode string,
     requestURI string,
     userAgent string,
     result string,
     resourceName string
   ) WITH (
     'connector' = 'aurora-sql'
   );
   ```

   This query uses the OpenX JSON SerDe (p. 147). The table format and the SerDe are suggested by the AWS Glue crawler when it analyzes AWS WAF logs.

   **Note**
   The SerDe expects each JSON record in the WAF logs in Amazon S3 to be on a single line of text with no line termination characters separating the fields in the record. If the WAF log JSON text is in pretty print format, you may receive the error message `HIVE_CURSOR_ERROR: Row is not a valid JSON Object when you attempt to query the table after you create it.`
CREATE EXTERNAL TABLE `waf_logs`(
    `timestamp` bigint,
    `formatversion` int,
    `webaclid` string,
    `terminatingruleid` string,
    `terminatingruletype` string,
    `action` string,
    `terminatingrulematchdetails` array<
        struct<
            conditiontype:string,
            location:string,
            matcheddata:array<string>
        >,
    `httpsourcename` string,
    `httpsourceid` string,
    `rulegrouplist` array<
        struct<
            rulegroupid:string,
            terminatingrule:struct<
                ruleid:string,
                action:string,
                rulematchdetails:string
            >,
            nonterminatingmatchingrules:array<
                struct<
                    ruleid:string,
                    action:string,
                    rulematchdetails:array<
                        struct<
                            conditiontype:string,
                            location:string,
                            matcheddata:array<string>
                        >
                    >
                >,
            excludedrules:array<
                struct<
                    ruleid:string,
                    exclusiontype:string
                >
            >
        >
    >
) `ratebasedrulelist` array<
    struct<
        ratebasedruleid:string,
        limitkey:string,
        maxrateallowed:int
    >
) `nonterminatingmatchingrules` array<
    struct<
        ruleid:string,
        action:string
    >,
) `requestheadersinserted` string,
) `responsecodesent` string,
) `httprequest` struct<
    clientip:string,
    country:string,
    297
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.

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 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 a single AWS region. In the LOCATION and storage.location.template clauses, replace the bucket and folder 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.

```
CREATE EXTERNAL TABLE `waf_logs`(
`timestamp` bigint,
`formatversion` int,
`webaclid` string,
`terminatingruleid` string,
`terminatingruletype` string,
`action` string,
`terminatingrulematchdetails` array<
  struct<
    conditiontype:string,
    location:string,
    matcheddata:array<string>
  >,
>,
`httpsourcename` string,
`httpsourceid` string,
`rulegrouplist` array<
  struct<
    `labels` array<
      struct<
        name:string
      >
    >,
    `headers` array<
      struct<
        name:string,
        value:string
      >,
    >,
    `uri`:string,
    `args`:string,
    `httpversion`:string,
    `httpmethod`:string,
    `requestid`:string
  >,
) ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'
WITH SERDEPROPERTIES ( 'paths'='action,formatVersion,httpRequest,httpSourceId,httpSourceName,labels,nonTerminatingMatchingRules,rateBasedRuleList ... rted,responseCodeSent,ruleGroupList,terminatingRuleId,terminatingRuleMatchDetails,terminatingRuleType,timestamp,webaclId')
STORED AS INPUTFORMAT 'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION 's3://athenawaflogs/WebACL/'
```
rulegroupid: string,
  terminatingrule: struct<
    ruleid: string,
    action: string,
    rulematchdetails: string
  >,
  nonterminatingmatchingrules: array<
    struct<
      ruleid: string,
      action: string,
      rulematchdetails: array<
        struct<
          conditiontype: string,
          location: string,
          matcheddata: array<string>
        >
      >
    >
  >
  >,
  excludedrules: array<
    struct<
      ruleid: string,
      exclusiontype: string
    >
  >,
  `ratebasedrulelist` array<
    struct<
      ratebasedruleid: string,
      limitkey: string,
      maxrateallowed: int
    >
  >,
  `nonterminatingmatchingrules` array<
    struct<
      ruleid: string,
      action: string
    >
  >,
  `requestheadersinserted` string,
  `responsecodesent` 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
    >
  >
PARTITIONED BY
(
  day 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/folder/

TBLPROPERTIES
(
"projection.enabled" = "true",
"projection.day.type" = "date",
"projection.day.range" = "2021/01/01,NOW",
"projection.day.format" = "yyyy/MM/dd",
"projection.day.interval" = "1",
"projection.day.interval.unit" = "DAYS",
"storage.location.template" = "s3://bucket/folder/${day}"
)

For more information about partition projection, see Partition Projection with Amazon Athena (p. 119).

Example Queries for AWS WAF Logs

In the following example queries, modify the table name, column values, and other variables 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 referers that contain a specified term

The following query counts the number of referers 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 day >= '2021/03/01'
  AND day < '2021/03/31')
SELECT COUNT(*) referer_count
FROM 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.

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.

```
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.

```
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.

```
SELECT * 
FROM waf_logs 
WHERE httprequest.clientip='53.21.198.66' AND day >= '2021/03/01' AND day < '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.

```
WITH test_dataset AS 
(SELECT 
    format_datetime(from_unixtime((timestamp/1000) - ((minute(from_unixtime(timestamp / 1000))%5) * 60)),'yyyy-MM-dd HH:mm') AS five_minutes_ts, 
    "httprequest"."clientip" 
    FROM waf_logs 
    WHERE day >= '2021/03/01' AND day < '2021/03/31') 
SELECT five_minutes_ts,"clientip",count(*) ip_count 
FROM test_dataset 
GROUP BY five_minutes_ts,"clientip"
```

For more information about date and time functions, see [Date and Time Functions and Operators](#) in the Presto 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.
SELECT COUNT(httpRequest.clientIp) as count,
    httpRequest.clientIp
FROM waf_logs
WHERE terminatingruletype='RATE_BASED' AND action='BLOCK' and day >= '2021/03/01'
AND day < '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.

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.

```sql
SELECT "httprequest"."clientip", "count"(*) "ipcount", "httprequest"."country"
FROM waf_logs
WHERE "action" = 'BLOCK' and day >= '2021/03/01'
    AND day < '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 Developer Guide
- Using AWS CloudTrail to identify Amazon S3 requests in the Amazon Simple Storage Service Developer Guide

### 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.

**Note**
You can use individual hive DDL commands (p. 502) to extract metadata information for specific databases, tables, views, partitions, and columns from Athena, but the output is in a non-tabular format.

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.

**Important**
You cannot use `CREATE VIEW` to create a view on the `information_schema` database.

**Topics**

- Listing Databases and Searching a Specified Database (p. 303)
- Listing Tables in a Specified Database and Searching for a Table by Name (p. 304)
- Listing Partitions for a Specific Table (p. 305)
- Listing or Searching Columns for a Specified Table or View (p. 306)

### 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;
```
Listing Tables in a Specified Database and Searching for a Table by Name

The following table shows sample results.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>alb-databases1</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
WHERE  schema_name = 'rdspostgresql'
```

The following table shows sample results.

<table>
<thead>
<tr>
<th>schema_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdspostgresql</td>
</tr>
</tbody>
</table>

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.

```sql
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>rdspostgresqldb1_public_acco</td>
<td>BASE TABLE</td>
</tr>
</tbody>
</table>

Example – Searching for a Table by Name

The following query obtains metadata information for the table athenal.
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 metadata query to list the partition numbers and partition values for a specific table. The syntax that you use depends on the Athena engine version.

**Example – Querying the Partitions for a Table in Athena engine version 2**

The following example query lists the partitions for the table `cloudtrail_logs_test2` using Athena engine version 2.

```
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>awsdatacatalog</td>
<td>cloudtrail_logs_test2</td>
<td>2020</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>awsdatacatalog</td>
<td>cloudtrail_logs_test2</td>
<td>2020</td>
<td>08</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>awsdatacatalog</td>
<td>cloudtrail_logs_test2</td>
<td>2020</td>
<td>08</td>
<td>12</td>
</tr>
</tbody>
</table>

**Example – Querying the Partitions for a Table in Athena engine version 1**

The following example query lists the partitions for the table `cloudtrail_logs_test2` using Athena engine version 1.

```
SELECT *
FROM   information_schema.__internal_partitions__
WHERE  table_schema = 'default'
AND     table_name = 'cloudtrail_logs_test2'
ORDER  BY partition_number
```

The following table shows sample results.
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 `rdspostgresql:rdspostgresqldb1_public_account`.

```
SELECT *
FROM   information_schema.columns
WHERE  table_schema = 'rdspostgresql'
AND table_name = 'rdspostgresqldb1_public_account'
```

The following table shows sample results.

<table>
<thead>
<tr>
<th>table_catalog</th>
<th>table_schema</th>
<th>table_name</th>
<th>partition_num</th>
<th>partition_key</th>
<th>partition_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>awsdatacatalog</td>
<td>default</td>
<td>cloudtrail_logs_test2</td>
<td>1</td>
<td>year</td>
<td>2018</td>
</tr>
<tr>
<td>awsdatacatalog</td>
<td>default</td>
<td>cloudtrail_logs_test2</td>
<td>1</td>
<td>month</td>
<td>09</td>
</tr>
<tr>
<td>awsdatacatalog</td>
<td>default</td>
<td>cloudtrail_logs_test2</td>
<td>1</td>
<td>day</td>
<td>30</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`.

```
SELECT *
FROM   information_schema.columns
WHERE  table_schema = 'default'
AND table_name = 'arrayview'
```

The following table shows sample results.
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>arrayviewsearchdate</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>arrayviewbtid</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>arrayviewmult</td>
<td>infantprice</td>
<td>3</td>
<td>YES</td>
<td>varchar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>awsdatacatalog</td>
<td>default</td>
<td>arrayviewmult</td>
<td>sump</td>
<td>5</td>
<td>YES</td>
<td>varchar</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

Topics

- Querying Apache Logs Stored in Amazon S3 (p. 307)
- Querying Internet Information Server (IIS) Logs Stored in Amazon S3 (p. 309)

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.
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. 143). 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 to point to your Apache logs in Amazon S3.

```sql
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.

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).

```
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.

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. 150) 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>IP Address</th>
<th>Request Target</th>
<th>Status Code</th>
<th>Bytes</th>
<th>Bytes</th>
<th>Time Taken</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>2020-01-19</td>
<td>22:48:39</td>
<td>203.0.113.5</td>
<td>/default.html</td>
<td>200</td>
<td>540</td>
<td>524</td>
<td>157</td>
<td>HTTP/1.0</td>
</tr>
<tr>
<td>2020-01-19</td>
<td>22:49:40</td>
<td>203.0.113.10</td>
<td>/index.html</td>
<td>200</td>
<td>420</td>
<td>324</td>
<td>164</td>
<td>HTTP/1.0</td>
</tr>
<tr>
<td>2020-01-19</td>
<td>22:50:12</td>
<td>203.0.113.12</td>
<td>/image.gif</td>
<td>200</td>
<td>324</td>
<td>320</td>
<td>358</td>
<td>HTTP/1.0</td>
</tr>
<tr>
<td>2020-01-19</td>
<td>22:51:44</td>
<td>203.0.113.15</td>
<td>/faq.html</td>
<td>200</td>
<td>330</td>
<td>324</td>
<td>288</td>
<td>HTTP/1.0</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. 106), 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.

```
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.

```
<table>
<thead>
<tr>
<th>date_col</th>
<th>time_col</th>
<th>cs_uri_stem</th>
<th>time_taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>2020-01-19</td>
<td>22:49:40</td>
<td>/index.html</td>
<td>164</td>
</tr>
<tr>
<td>2020-01-19</td>
<td>22:50:12</td>
<td>/image.gif</td>
<td>358</td>
</tr>
<tr>
<td>2020-01-19</td>
<td>22:51:44</td>
<td>/faq.html</td>
<td>288</td>
</tr>
</tbody>
</table>
```

### 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. 474) or `CREATE TABLE AS SELECT` (p. 514) 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.

```sql
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.

```sql
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.
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 Log Files

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,
user_name string,
request_date string,
request_time string,
service_and_instance string,
server_name string,
service_ip_address string,
time_taken_millisec string,
client_bytes_sent string,
server_bytes_sent string,
)
```
service_status_code string,
windows_status_code string,
request_type string,
target_of_operation string,
script_parameters 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.

![Results](image)

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.

The following example shows data in the NCSA common log format as documented for IIS.

```
198.51.100.7 - ExampleCorp\Li [10/Oct/2019:13:55:36 -0700] "GET /logo.gif HTTP/1.0" 200 232
198.51.100.22 - ExampleCorp\Mateo [27/Dec/2019:11:38:12 -0700] "GET /about.html HTTP/1.1" 200 1287
```
Creating a Table in Athena for IIS NCSA Logs

For your CREATE TABLE statement, you can use the Grok SerDe (p. 143) and a grok pattern similar to the one for Apache web server logs (p. 307). 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.

```
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.
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. 316)
- Identity and Access Management in Athena (p. 324)
- Logging and Monitoring in Athena (p. 366)
- Compliance Validation for Amazon Athena (p. 369)
- Resilience in Athena (p. 370)
- Infrastructure Security in Athena (p. 370)
- Configuration and Vulnerability Analysis in Athena (p. 372)
- Using Athena to Query Data Registered With AWS Lake Formation (p. 372)

**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.

## 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.

1. **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 Developer 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. 321).

2. **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. 34).

You can define fine-grained access policies to databases and tables (p. 335) 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. 320) for access.

3. **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 Query Results Stored in Amazon S3 (p. 320) in this document.

   Athena retains query history for 45 days. You can view query history (p. 170) 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. 420) in Athena, restricting access to saved queries only to users who are authorized to view them.

### Topics
- Encryption at Rest (p. 318)
• Encryption in Transit (p. 323)
• Key Management (p. 324)
• Internetwork Traffic Privacy (p. 324)

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. 320).

• The data in the AWS Glue Data Catalog. For information, see Permissions to Encrypted Metadata in the AWS Glue Data Catalog (p. 320).

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. 318)
• Permissions to Encrypted Data in Amazon S3 (p. 319)
• Permissions to Encrypted Metadata in the AWS Glue Data Catalog (p. 320)
• Encrypting Athena Query Results Stored in Amazon S3 (p. 320)
• Creating Tables Based on Encrypted Datasets in Amazon S3 (p. 321)

Supported Amazon S3 Encryption Options

Athena supports the following encryption options for datasets and query results in Amazon S3. Cross-regional support is limited to certain Regions. For more information, see Cross-Regional Queries (p. 529).

<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>

Note
With this encryption type, Athena does not require you to indicate that data is encrypted when you create a table.
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 master key.
- Asymmetric keys.

To compare Amazon S3 encryption options, see Protecting Data Using Encryption in the Amazon Simple Storage Service Developer 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 IAM Policies for User Access (p. 325) and Amazon S3 Permissions (p. 331).
- **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. In this case, you can 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. 535) section of the Troubleshooting in Athena (p. 530) 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 to Encrypted Metadata in the AWS Glue Data Catalog (p. 344).

**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. 430) and select the Override client-side settings field, then all queries that run in this workgroup use the workgroup settings. For more information, see Workgroup Settings Override Client-Side Settings (p. 429). Workgroup settings include query results location and encryption.

**To encrypt query results stored in Amazon S3 using the console**

**Important**
If your workgroup has the Override client-side settings field selected, then the queries use the workgroup settings. The encryption configuration and the query results location listed in Settings, the API operations, and the drivers are not used. For more information, see Workgroup Settings Override Client-Side Settings (p. 429).

1. In the Athena console, choose Settings.

2. For Query result location, enter a custom value or leave the default. This is the Amazon S3 staging directory where query results are stored.
3. Choose **Encrypt query results**.

4. For **Encryption type**, choose **CSE-KMS**, **SSE-KMS**, or **SSE-S3**.

5. If you chose **SSE-KMS** or **CSE-KMS**, specify the **Encryption key**.
   
   - If your account has access to an existing AWS KMS customer managed key (CMK), choose its alias or choose **Enter a KMS key ARN** and then enter an ARN.
   
   - If your account does not have access to an existing AWS KMS customer managed key (CMK), choose **Create KMS key**, and then open the **AWS KMS console**. In the navigation pane, choose **AWS managed keys**. 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.

6. Return to the Athena console to specify the key by alias or ARN as described in the previous step.

7. 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. 86)**.

### 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. 511) 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. 87) and set the **TBLPROPERTIES** value as shown in the previous example when you use `statement.executeQuery()` to run the **CREATE TABLE** (p. 511) statement.

• When you use the Athena console **Add table wizard** (p. 102) and specify a table location, select the **Encrypted data set** option.

In the Athena console **Tables** list, the encrypted table shows an encryption 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 Connect with the JDBC Driver (p. 87) and Connect with the ODBC Driver (p. 88).

**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 master 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 Developer Guide**.

If your Athena workloads encrypt a large amount of data, you can use 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 on Amazon S3 bucket IAM policies.

**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**.

---

324
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 Developer 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. 325)
- Access through JDBC and ODBC Connections (p. 330)
- Access to Amazon S3 (p. 331)
- Cross-account Access in Athena to Amazon S3 Buckets (p. 331)
- Fine-Grained Access to Databases and Tables in the AWS Glue Data Catalog (p. 335)
- Cross-Account Access to Data Catalogs (p. 341)
- Access from Athena to Encrypted Metadata in the AWS Glue Data Catalog (p. 344)
- Access to Workgroups and Tags (p. 345)
- Allow Access to Prepared Statements (p. 345)
- Using Athena with CalledVia Context Keys (p. 346)
- Allow Access to an Athena Data Connector for External Hive Metastore (p. 348)
- Allow Lambda Function Access to External Hive Metastores (p. 350)
- Example IAM Permissions Policies to Allow Athena Federated Query (p. 354)
- Example IAM Permissions Policies to Allow Amazon Athena User Defined Functions (UDF) (p. 358)
- Allowing Access for ML with Athena (p. 362)
- Enabling Federated Access to the Athena API (p. 362)

### 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 Read only Access AWS managed policy provides read-only access to all 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 Service Actions for JDBC Connections (p. 330).
- 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 to Encrypted Metadata in the AWS Glue Data Catalog (p. 344).
- 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. 423) and the section called “Workgroup Example Policies” (p. 424).

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. 437).
- **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": "arn:aws:athena:us-east-1:::query/<query_id>",
            "Condition": {
                "StringEquals": {
                    "athena:WorkGroup": "<workgroup_name>",
                    "athena:Database": "<database_name>",
                    "athena:Table": "<table_name>"
                }
            }
        }
    ]
}
```
"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",
               "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:PutObject"
  ],
  "Resource": [  "arn:aws:s3:::aws-athena-query-results-*"
]
}
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.
"Version": "2012-10-17",
"Statement": [
  {
    "Effect": "Allow",
    "Action": [
      "athena:BatchGetQueryExecution",
      "athena:GetQueryExecution",
      "athena:GetQueryResults",
      "athena:GetQueryResultsStream",
      "athena:ListQueryExecutions",
      "athena:StartQueryExecution",
      "athena:GetQueryExecution",
      "athena:GetDataCatalog",
      "athena:GetWorkGroup",
      "athena:GetEngineVersions",
      "athena:GetWorkGroup",
      "athena:GetDatabase",
      "athena:GetTableMetadata",
      "athena:GetDatabase",
      "athena:GetTableMetadata"
    ],
    "Resource": ["*"]
  },
  {
    "Effect": "Allow",
    "Action": [
      "glue:CreateDatabase",
      "glue:DeleteDatabase",
      "glue:GetDatabase",
      "glue:GetDatabases",
      "glue:UpdateDatabase",
      "glue:GetTable",
      "glue:DeleteTable",
      "glue:BatchDeleteTable",
      "glue:GetTable`
      "glue:GetTable`
      "glue:GetTables",
      "glue:BatchCreatePartition",
      "glue:CreatePartition",
      "glue:DeletePartition",
      "glue:BatchDeletePartition",
      "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",
      "s3:PutBucketPublicAccessBlock"
    ]
  }
]
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>AmazonAthenaFullAccess (p. 326) and AWSQuicksightAthenaAccess (p. 328) – Updates to existing policies</td>
<td>Athena added s3:PutBucketPublicAccessBlock to enable the blocking of public access on the buckets created by Athena.</td>
<td>July 7, 2021</td>
</tr>
<tr>
<td>Athena started tracking changes for its AWS managed policies.</td>
<td>Athena started tracking changes for its AWS managed policies.</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 AWSQuicksightAthenaAccess Managed Policy (p. 328).

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. 87) and ODBC (p. 88) driver pages. For additional related information, see the following:
• Enabling Federated Access to the Athena API (p. 362)
• Using Lake Formation and the Athena JDBC and ODBC Drivers for Federated Access to Athena (p. 380)

For information about the latest versions of the JDBC and ODBC drivers and their documentation, see Using Athena with the JDBC Driver (p. 87) and Connecting to Amazon Athena with ODBC (p. 88).

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. 109) and Using access points in the Amazon S3 User Guide.

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. 331).

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 Data Catalogs (p. 341).
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",
            "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:
  ```json
  {
  }
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.

   ```json
   {
     "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": "*"
       }
     ]
   }
   ```

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.

   ```json
   "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:

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ExampleStmt2",
            "Action": [
                "s3:GetObject",
                "s3:PutObject"
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:s3:::awsexamplebucket/*"
        }
    ]
}
```

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.

```json
{
    "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 following Data Catalog objects that are used in Athena: databases and tables.

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. 335)
- Mandatory: Access Policy to the Default Database and Catalog per AWS Region (p. 336)
- Table Partitions and Versions in AWS Glue (p. 337)
- Fine-Grained Policy Examples (p. 337)

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. 337).
- 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 Fine-Grained Policy Examples (p. 337).

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 Fine-Grained Policy Examples (p. 337).

• 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 descendants. A table does not have descendants. To run DROP TABLE, you need permissions to this action on the table, the database to which it belongs, and the catalog. See Fine-Grained Policy Examples (p. 337).

• 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 the 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 two actions: GetDatabase and CreateDatabase:

```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/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 and CreateDatabase must be present:

```json
{
  "Effect": "Allow",
  "Action": [
    "glue:GetDatabase",
    "glue:CreateDatabase"
  ],
  "Resource": [
```
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](https://docs.aws.amazon.com/glue/latest/dg/permissions-reference.html) 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>CREATE DATABASE</td>
<td>Allows you to create the database named <code>example_db</code>.</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/example_db&quot;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
### DDL Statement

<table>
<thead>
<tr>
<th>Example of an IAM access policy granting access to the resource</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ALTER DATABASE</code> Allows you to modify the properties for the example_db database.</td>
</tr>
</tbody>
</table>

```json

```

<table>
<thead>
<tr>
<th>Example of an IAM access policy granting access to the resource</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DROP DATABASE</code> Allows you to drop the example_db database, including all tables in it.</td>
</tr>
</tbody>
</table>

```json

```
<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><strong>SHOW DATABASES</strong></td>
<td>Allows you to list all databases in the AWS Glue Data Catalog.</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:GetDatabases&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/*&quot;</td>
</tr>
<tr>
<td></td>
<td>]</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td><strong>CREATE TABLE</strong></td>
<td>Allows you to create a table named test in the example_db database.</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: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:GetTable&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;glue:CreateTable&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/test&quot;</td>
</tr>
<tr>
<td></td>
<td>]</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
### DDL Statement

<table>
<thead>
<tr>
<th>SHOW TABLES</th>
<th>Allows you to list all tables in the example_db database.</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>
|             |       "arn:aws:glue:us-east-1:123456789012:database/example_db",
|             |       "arn:aws:glue:us-east-1:123456789012:table/example_db/****
|             |     ]                                                    |
|             | }                                                        |
| DROP TABLE  | 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. |
|             | {                                                         |
|             |   "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"
|             |     ]                                                    |
|             | }                                                        |
Cross-Account Access to 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 on using the Athena console to register a catalog from another account, see Registering an AWS Glue Data Catalog from Another Account (p. 22).

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. 18) - Contains topics on using Athena with AWS Glue, Hive, or Lambda data catalog sources.
- Data Catalog Example Policies (p. 81) - Shows how to write policies that control access to data catalogs.
- Using the AWS CLI with Hive Metastores (p. 61) - 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:

- The feature is available only in Regions where Athena engine version 2 is supported. For a list of AWS Regions that support Athena engine version 2, see Athena engine version 2 (p. 463). To upgrade a workgroup to engine version 2, see Changing Athena Engine Versions (p. 459).
- 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:

```sql
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.

---

341
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 Creates an Athena Data Catalog that Points 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. 81).

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.

**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:

```sql
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 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. 520) 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 Console 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:

```
SELECT *
```

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 (``).

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:

```
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. 424).
• For a list of tag-based policies for workgroups, see Tag-Based IAM Access Control Policies (p. 453).
• For more information about creating IAM policies for workgroups, see Workgroup IAM Policies (p. 423).
• 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.

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.

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.

For more information about prepared statements, see Querying with Prepared Statements (p. 195).

The following IAM permissions are required for creating, managing, and executing prepared statements.

- `athena:CreatePreparedStatement`
- `athena:UpdatePreparedStatement`
- `athena:GetPreparedStatement`
Use these permissions as shown in the following table.

<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>Run a PREPARE query</td>
<td>athena:CreatePreparedStatement</td>
</tr>
<tr>
<td>Run a PREPARE query</td>
<td>athena:UpdatePreparedStatement</td>
</tr>
<tr>
<td>Run an EXECUTE query</td>
<td>athena:StartQueryExecution</td>
</tr>
<tr>
<td>Run a EXECUTE query</td>
<td>athena:GetPreparedStatement</td>
</tr>
<tr>
<td>Run a DEALLOCATE  PREPARE query</td>
<td>athena:StartQueryExecution</td>
</tr>
<tr>
<td>Run a DEALLOCATE  PREPARE query</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",
        "athena:GetPreparedStatement",
        "athena:DeletePreparedStatement",
        "athena:ListPreparedStatements"
      ],
      "Resource": [
        "arn:aws:athena:*:11112223333:workgroup/<workgroup-name>"
      ]
    }
  ]
}
```

**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
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Sid": "VisualEditor3",  
      "Effect": "Allow",  
      "Action": "lambda:InvokeFunction",  
      "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.

```json
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Sid": "VisualEditor0",  
      "Effect": "Allow",  
      "Action": [  
        "athena:GetWorkGroup",  
        "s3:PutObject",
```
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. 349)
- Example Policy to Allow an IAM Principal to Create an Athena Data Connector for External Hive Metastore (p. 350)

For more information about CalledVia condition keys, see AWS global condition context keys in the IAM User Guide.
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. 326), 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>
</table>
| "s3:GetBucketLocation",
"s3:GetObject",
"s3:ListBucket",
"s3:PutObject",
"s3:ListMultipartUploadParts",
"s3:AbortMultipartUpload" | s3 actions allow reading from and writing to the resource specified as "arn:aws:s3:::MyLambdaSpillBucket/MyLambdaSpillLocation", where MyLambdaSpillLocation identifies the spill bucket that is specified in the configuration of the Lambda function or functions being invoked. The arn:aws:lambda::*:MyAWSAcctId:layer:MyAthenaLambdaLayer:arn resource identifier is required only if you use a Lambda layer to create custom runtime dependencies to reduce function artifact size at deployment time. The * in the last position is a wildcard for layer version. |
| "lambda:GetFunction",
"lambda:GetLayerVersion",
"lambda:InvokeFunction"          | Allows queries to invoke the AWS Lambda functions specified in the Resource block. For example, arn:aws:lambda::*:MyAWSAcctId:function:MyAthenaLambdaFunction |
Allow Lambda Function Access to External Hive Metastores

<table>
<thead>
<tr>
<th>Allowed Actions</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>where <em>MyAthenaLambdaFunction</em> 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 Data Connector for External Hive Metastore

The following policy is attached to IAM principals in addition to the AWS managed policy: AmazonAthenaFullAccess (p. 326), 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:*:MyAthenaLambdaFunctionPrefix"
      }
   ]
}
```

Explanation of Permissions

Allows queries to invoke the AWS Lambda functions for the AWS Lambda functions specified in the Resource block. For example, `arn:aws:lambda:*:MyAWSActId:function:*:MyAthenaLambdaFunction`, where *MyAthenaLambdaFunction* specifies the name of a Lambda function to be invoked. Multiple functions can be specified as shown in the example.

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. 326) 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.

```
{
   "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. 354).

Catalog Registration and Metadata API Operations

For access to catalog registration API and metadata API operations, use the AmazonAthenaFullAccess managed policy (p. 326). If you do not use this policy, add the following API operations to your Athena policies:

```
{
   "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 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": "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
```

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. 352) 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.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "Stmt1234567890123",
         "Effect": "Allow",
         "Principal": {
            "AWS": "arn:aws:iam::444455556666:user/perf1-test"
         },
         "Action": "s3:GetObject",
         "Resource": "arn:aws:s3:::athena-test/lambda/dataset/*"
      }
   ]
}
```
Note
You might need to grant cross account access to Amazon S3 not only to your data, but also to your Amazon S3 spill location. Your Lambda function spills extra data to the spill location when the size of the response object exceeds a given threshold. See the beginning of this topic for a sample policy.

In the current example, after cross account access is granted to 444455556666, 444455556666 can use catalog ehms in its own account to query tables that are defined in account 111122223333.

In the following example, the SQL Workbench profile perf-test-1 is for account 444455556666. The query uses catalog ehms to access the Hive metastore and the Amazon S3 data in account 111122223333.

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. 354)
- Example Policy to Allow an IAM Principal to Create a Data Source Connector (p. 356)

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.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": [
                "athena:GetWorkGroup",
                "s3:PutObject",
                "s3:GetObject",
                "athena:StartQueryExecution",
                "s3:AbortMultipartUpload"
            ]
        }
    ]
}
```
Amazon Athena User Guide
Allow Access to Athena Federated Query

```
"lambda:InvokeFunction",
"athena:StartQueryExecution",
"athena:GetQueryExecution",
"athena:GetQueryResults",
"athena:GetWorkGroup",
"athena:StopQueryExecution",
"s3:ListMultipartUploadParts"
],
"Resource": [
  "arn:aws:athena:*:111122223333:workgroup/WorkgroupName",
  "arn:aws:s3:::MyQueryResultsBucket/*",
  "arn:aws:s3:::MyLambdaSpillBucket/MyLambdaSpillPrefix*",
  "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>
<tbody>
<tr>
<td>&quot;athena:StartQueryExecution&quot;, &quot;athena:GetQueryResults&quot;, &quot;athena:GetWorkGroup&quot;, &quot;athena:StopQueryExecution&quot;, &quot;athena:GetQueryExecution&quot;,</td>
<td>Athena permissions that are required to run federated queries.</td>
</tr>
<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:::MyQueryResultsBucket/* resource identifier, where MyQueryResultsBucket is the Athena query results bucket. For more information, see Working with Query Results, Output Files, and Query History (p. 162). s3:GetObject 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. s3:GetObject also allows reading from the resource specified as</td>
</tr>
</tbody>
</table>
## Allowed Actions

<table>
<thead>
<tr>
<th>Action</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;arn:aws:s3:::MyLambdaSpillBucket/MyLambdaSpillPrefix*&quot;</td>
<td>where <strong>MyLambdaSpillPrefix</strong> 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 <strong>Resource</strong> block. For example, <strong>arn:aws:lambda::MyAWSAcctId:function:MyAthenaLambdaFunction</strong> where <strong>MyAthenaLambdaFunction</strong> 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: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:GetFunctionAlias",
        "lambda:GetFunctionConfiguration",
        "iam:DeleteRole",
        "lambda:UpdateFunctionConfiguration",
        "s3:GetObject",
        "lambda:AddPermission",
        "iam:UpdateRole",
        "lambda:DeleteFunctionConcurrency",
        "lambda:RemovePermission",
        "iam:GetRolePolicy",
        "lambda:GetPolicy"
      ],
      "Resource": [
        "arn:aws:lambda::MyAWSAcctId:function:MyAthenaLambdaFunctionsPrefix*",
        "arn:aws:s3:::awsserverlessrepo-changesets-l1iv3xa62ln3m/**",
        "arn:aws:iam::*:role/**",
        "arn:aws:iam::MyAWSAcctId:policy/**"
      ]
    }
  ]
}```
Amazon Athena User Guide
Allow Access to Athena Federated Query

```
{
  "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:*:MyAWSAcctId:stack/aws-serverless-repository-MyCFStackPrefix*/*",
    "arn:aws:cloudformation:*:MyAWSAcctId:stack/serverlessrepo-MyCFStackPrefix*/*",
    "arn:aws:cloudformation:*:transform/Serverless-**",
    "arn:aws:cloudformation:*:MyAWSAcctId:stackset/aws-serverless-repository-MyCFStackPrefix*/*",
    "arn:aws:cloudformation:*:MyAWSAcctId: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;</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 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>
**Allow Access to Athena UDF**

**Allowed Actions**

<table>
<thead>
<tr>
<th>Action</th>
<th>Explanation</th>
</tr>
</thead>
</table>
| "lambda:UpdateFunctionConfiguration"        | Allows reading of a bucket that AWS Serverless Application Repository requires as specified by the resource identifier arn:aws:s3:::awsserverlessrepo-changesets-1iiiv3xa62ln3m/*.
| "lambda:UpdateFunctionCode"                 | This bucket may be specific to your account.                                |
| "lambda:AddPermission"                       |                                                                             |
| "lambda:DeleteFunctionConcurrency"          |                                                                             |
| "lambda:RemovePermission"                   |                                                                             |
| "lambda:GetPolicy"                           |                                                                             |
| "lambda:GetAccountSettings"                  |                                                                             |
| "lambda:ListFunctions"                       |                                                                             |
| "lambda:ListEventSourceMappings"             |                                                                             |
| "s3:GetObject"                               | Allows reading of a bucket that AWS Serverless Application Repository requires as specified by the resource identifier arn:aws:s3:::awsserverlessrepo-changesets-1iiiv3xa62ln3m/*.
| "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. 358)
- Example Policy to Allow an IAM Principal to Create an Athena UDF (p. 360)

**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",
        }
    ]
}
```
Amazon Athena User Guide
Allow Access to Athena UDF

```
"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": "*
}
```
### Allowed Actions

<table>
<thead>
<tr>
<th>MyLambdaSpillPrefix is specified in the configuration of the Lambda function or functions being invoked.</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;lambda:InvokeFunction&quot; Allows queries to invoke the AWS Lambda functions specified in the Resource block. For example, arn:aws:lambda::*:MyAWSAcutId:fnction: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:GetTags",
                "iam:GetRolePolicy",
                "lambda:GetPolicy",
                "lambda:InvokeFunction",
                "lambda:GetFunction",
                "lambda:ListAliases",
                "lambda:UpdateFunctionConfiguration",
                "iam:DeleteRole",
                "lambda:DeleteFunction",
                "lambda:UpdateFunctionCode",
                "s3:GetObject",
                "lambda:AddPermission",
                "iam:UpdateRole",
                "lambda:DeleteFunctionConcurrency",
                "lambda:RemovePermission",
                "iam:GetRolePolicy",
                "lambda:GetPolicy"
            ],
            "Resource": [
                "arn:aws:lambda::*:MyAWSAcutId:fnction:MyAthenaLambdaFunctionsPrefix*",
                "arn:aws:s3:::awsserverlessrepo-changesets-iiivjx62ln3m/*",
                "arn:aws:iam::*:rolename/",
                "arn:aws:iam::MyAWSAcutId:policy/*"
            ]
        }
    ]
}
```
Amazon Athena User Guide
Allow Access to Athena UDF

```
{
   "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:*:MyAWSAcctId:stack/aws-serverless-repository-MyCFStackPrefix/*",
               "arn:aws:cloudformation:*:MyAWSAcctId:stack/serverlessrepo-MyCFStackPrefix/*",
               "arn:aws:cloudformation:*::*:transform/Serverless-*",
               "arn:aws:cloudformation:*:MyAWSAcctId:stackset/aws-serverless-repository-MyCFStackPrefix/*",
               "arn:aws:cloudformation:*:MyAWSAcctId:stackset/serverlessrepo-MyCFStackPrefix/*"
               ],
   "Resource": ["arn:aws:cloudformation:*:MyAWSAcctId:stack/aws-serverless-repository-MyCFStackPrefix/*",
               "arn:aws:cloudformation:*:MyAWSAcctId:stack/serverlessrepo-MyCFStackPrefix/*",
               "arn:aws:cloudformation:*::*:transform/Serverless-*",
               "arn:aws:cloudformation:*:MyAWSAcctId:stackset/aws-serverless-repository-MyCFStackPrefix/*",
               "arn:aws:cloudformation:*:MyAWSAcctId:stackset/serverlessrepo-MyCFStackPrefix/*"
               ],
   "Resource": ["arn:aws:serverlessrepo:*:*:applications/*"]
}
```

### Explanation of Permissions

<table>
<thead>
<tr>
<th>Allowed Actions</th>
<th>Explanation</th>
</tr>
</thead>
</table>
| "lambda:CreateFunction",
"lambda:ListVersionsByFunction",
"lambda:GetFunctionConfiguration",
"lambda:PutFunctionConcurrency",
"lambda:GetTags",
"lambda:DeleteFunction",
"lambda:GetAlias",
"lambda:InvokeFunction",
"lambda:GetFunction",
"lambda:GetAliases",
"lambda:UpdateFunctionConfiguration",
"lambda:UpdateFunctionCode" | 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 `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. |
# 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. 326), 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.

```
{
  "Effect": "Allow",
  "Action": [
    "sagemaker:invokeEndpoint"
  ],
default"
}
```

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.

## 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.

### Allowed Actions

<table>
<thead>
<tr>
<th>Action</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;lambda:AddPermission&quot;</td>
<td>Allows reading of a bucket that AWS Serverless Application Repository requires as specified by the resource identifier <code>arn:aws:s3:::awsserverlessrepo-changesets-11iv3xa62ln3m/*</code>.</td>
</tr>
<tr>
<td>&quot;s3:GetObject&quot;</td>
<td>Allows reading of a bucket that AWS Serverless Application Repository requires as specified by the resource identifier <code>arn:aws:s3:::awsserverlessrepo-changesets-11iv3xa62ln3m/*</code>.</td>
</tr>
<tr>
<td>&quot;cloudformation:*&quot;</td>
<td>Allows the creation and management of AWS CloudFormation stacks specified by the resource identifier <code>arn:aws:cloudformation:*:*:stacks/MyCFStackPrefix</code>. These stacks and stacksets are how AWS Serverless Application Repository deploys connectors and UDFs.</td>
</tr>
<tr>
<td>&quot;serverlessrepo:*&quot;</td>
<td>Allows searching, viewing, publishing, and updating applications in the AWS Serverless Application Repository, specified by the resource identifier <code>arn:aws:serverlessrepo:*:*:applications/*</code>.</td>
</tr>
</tbody>
</table>

---

362
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. 87) and Connecting to Amazon Athena with ODBC (p. 88).

**Topics**
- Before You Begin (p. 363)
- Architecture Diagram (p. 363)
- Procedure: SAML-based Federated Access to the Athena API (p. 364)

**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. 87) and Connecting to Amazon Athena with ODBC (p. 88).

**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. 87) and Connecting to Amazon Athena with ODBC (p. 88).

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. 87) and Connecting to Amazon Athena with ODBC (p. 88) 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. 363).
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 programatically.

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. 367).

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. 276) and CloudTrail SerDe (p. 138).

- **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. 422). Use resource-level IAM permissions to control access to a specific workgroup. For more information, see Using Workgroups for Running Queries (p. 420) and Controlling Costs and Monitoring Queries with CloudWatch Metrics and Events (p. 437).

**Topics**

- Logging Amazon Athena API Calls with AWS CloudTrail (p. 367)
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. 276), CloudTrail SerDe (p. 138), 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. 368)
- **StartQueryExecution (Failed)** (p. 368)
- **CreateNamedQuery** (p. 369)

### StartQueryExecution (Successful)

```
{
  "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",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```

### StartQueryExecution (Failed)

```
{
  "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",
  "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. 325) 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. 331).

Topics
- Connect to Amazon Athena Using an Interface VPC Endpoint (p. 371)

Connect to Amazon Athena Using an Interface VPC Endpoint

You can connect directly to Athena by using an interface VPC endpoint (AWS PrivateLink) and an AWS Glue VPC endpoint in your Virtual Private Cloud (VPC) instead of connecting over the internet. When you use an interface VPC endpoint, communication between your VPC and Athena is conducted entirely within the AWS network. 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
Amazon Athena User Guide
How Lake Formation Data Access Works

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. 373)
• Considerations and Limitations When Using Athena to Query Data Registered With Lake Formation (p. 375)
• Managing Lake Formation and Athena User Permissions (p. 377)
• Applying Lake Formation Permissions to Existing Databases and Tables (p. 379)
• Using Lake Formation and the Athena JDBC and ODBC Drivers for Federated Access to Athena (p. 380)

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. 375)
- Working With Lake Formation Permissions To Views (p. 375)
- Athena Query Results Location In Amazon S3 Not Registered With Lake Formation (p. 375)
- Use Athena Workgroups To Limit Access To Query History (p. 376)
- Cross-Account Data Catalog Access (p. 376)
- CSE-KMS Encrypted Amazon S3 Locations Registered With Lake Formation Cannot Be Queried in Athena (p. 376)
- Partitioned Data Locations Registered with Lake Formation Must Be in Table Subdirectories (p. 376)
- Create Table As Select (CTAS) Queries Require Amazon S3 Write Permissions (p. 377)
- The DESCRIBE Permission is Required on the Default Database (p. 377)

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 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.

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, Output Files, and Query History (p. 162).

**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. 420).

**Cross-Account Data Catalog Access**

To access a data catalog in another account, you can use one of the following methods:

- 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 is limited to same-Region use between accounts. For more information, see Registering an AWS Glue Data Catalog from Another Account (p. 22) and Cross-Account Access to Data Catalogs (p. 341).

- Set up cross-account access in Lake Formation. For more information, see the following section.

**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 more information, 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. 177).

**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.

```bash
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. 378)
- Amazon S3 Permissions For Athena Query Results Locations (p. 378)
- Athena Workgroup Memberships To Query History (p. 378)
- Lake Formation Permissions To Data (p. 378)
- IAM Permissions to Write to Amazon S3 Locations (p. 378)
- Permissions to Encrypted Data, Metadata, and Athena Query Results (p. 379)
- Resource-Based Permissions for Amazon S3 Buckets in External Accounts (Optional) (p. 379)
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` (p. 326) 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. 420), IAM Policies for Accessing Workgroups (p. 423), and Enabling Federated Access to the Athena API (p. 362).

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, Output Files, and Query History (p. 162).

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. 420).

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. 177).

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 Query Results Stored in Amazon S3 (p. 320).

- **Encrypting source data** – SSE-S3 and CSE-KMS encryption of Amazon S3 data locations source data
  is supported. SSE-KMS encryption is not 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 Permissions to Encrypted Data in Amazon S3 (p. 319).

- **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 to Encrypted Metadata in the AWS Glue
  Data Catalog (p. 344).

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. 331).

For information about accessing a Data Catalog in another account, see Cross-Account Data Catalog
Access (p. 376).

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. 335) 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. 381).

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. 88).

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. 381)
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. 167) 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. 97) 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. 381)
- Step 2: Add users and groups to Okta (p. 382)
- Step 3: Set up an Okta Application for SAML Authentication (p. 389)
- Step 4: Create an AWS SAML Identity Provider and Lake Formation Access IAM Role (p. 396)
- Step 5: Add the IAM Role and SAML Identity Provider to the Okta Application (p. 400)
- Step 6: Grant user and group permissions through AWS Lake Formation (p. 406)
- Step 7: Verify access through the Athena JDBC client (p. 410)
- Conclusion (p. 419)
- Related Resources (p. 419)

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.
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.

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.

2. Choose **Manage People**.
3. From the **Not Members** list, choose **athena-okta-user**.

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**.

![Create role](image)
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.

   ```json
   {
   "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.

   ```json
   {
   "Version": "2012-10-17",
   "Statement": [
   {
   "Sid": "AthenaQueryResultsPermissionsForS3",
   "Effect": "Allow",
   "Action": [
   "s3:ListBucket",
   "s3:PutObject",
   "s3:GetObject"
   ],
   "Resource": "*"
   }
   }
   ```
4. Choose **Review policy**.
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**.
4. 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**
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:

   \[
   \text{arn:aws:iam::<account-id>:saml-provider/AthenaLakeFormationOkta:user/\langle\text{athena-okta-user}\rangle@<\text{anycompany.com}>}
   \]

   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:
   ```plaintext
   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.
The Data permissions page for the `nyctaxi` table shows the permissions for `athena-okta-user` and the `lf-business-analyst` group.

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. 87).
2. Download and install the free SQL Workbench/J SQL query tool, available under a modified Apache 2.0 license.
3. In SQL Workbench, choose File, and then choose Manage Drivers.

   ![SQL Workbench Menu]

   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. 87). 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"
   ```

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.

```sql
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;
pwd=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.

```ini
[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.

```sql
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;
pwd=password;
idp_host=okta-idp-domain;
App_ID=okta-application-id;
SSL_Insecure=true;
LakeFormationEnabled=true;
```
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.

```
1 describe 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></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>trip_pickup_datetime</td>
<td>timestamp</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>trip_dropoff_datetime</td>
<td>timestamp</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td></td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>
```

```
1 select * from tripdb.nyctaxi limit 5 |
```

```
<table>
<thead>
<tr>
<th>vendor</th>
<th>trip_pickup_datetime</th>
<th>trip_dropoff_datetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017-01-01 00:01:15</td>
<td>2017-01-01 00:11:05</td>
<td></td>
</tr>
<tr>
<td>2017-01-01 00:03:34</td>
<td>2017-01-01 00:09:00</td>
<td></td>
</tr>
<tr>
<td>2017-01-01 00:04:02</td>
<td>2017-01-01 00:12:35</td>
<td></td>
</tr>
<tr>
<td>2017-01-01 00:01:40</td>
<td>2017-01-01 00:14:23</td>
<td></td>
</tr>
<tr>
<td>2017-01-01 00:00:51</td>
<td>2017-01-01 00:18:35</td>
<td></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.

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.

   ![SQL Workbench Screenshot](image)

   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. 87)
- Enabling Federated Access to the Athena API (p. 362)
- 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, and Amazon Simple Notification Service 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.

Topics
- Using Workgroups for Running Queries (p. 420)
- Controlling Costs and Monitoring Queries with CloudWatch Metrics and Events (p. 437)

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. 421)
- How Workgroups Work (p. 421)
- Setting up Workgroups (p. 422)
- IAM Policies for Accessing Workgroups (p. 423)
- Workgroup Settings (p. 428)
- Managing Workgroups (p. 430)
- Athena Workgroup APIs (p. 435)
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. 421). For all queries in the workgroup, you can choose to configure workgroup settings. They include an Amazon S3 location for storing query results, and encryption configuration. You can also enforce workgroup settings. For more information, see Workgroup Settings (p. 428).</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>
<tr>
<td></td>
<td>• <strong>Per-query limit</strong> 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.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Per-workgroup limit</strong> 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. 444).</td>
</tr>
<tr>
<td>Track query-related metrics for all workgroup queries in CloudWatch.</td>
<td>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. 438) for each of your workgroups within the Athena console. In CloudWatch, you can create custom dashboards, and set thresholds and alarms on these metrics.</td>
</tr>
</tbody>
</table>

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 and encryption configuration.
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. 429).

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. 434) 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:

Workgroups display in the Athena console in the Workgroup:<workgroup_name> tab. The console lists the workgroup that you have switched to. When you run queries, they run in this workgroup. You can run queries in the workgroup in the console, or by using the API operations, the command line interface, or a client application through the JDBC or ODBC driver. When you have access to a workgroup, you can view workgroup's settings, metrics, and data usage control limits. Additionally, you can have permissions to 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. 423).
   - 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. 428).
   - 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 Query Results Stored in Amazon S3 (p. 320).
2. Create workgroups as needed, and add tags to them. Open the Athena console, choose the Workgroup:<workgroup_name> tab, and then choose Create workgroup. For detailed steps, see Create a Workgroup (p. 430).

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. 423). For example JSON policies, see Workgroup Example Policies (p. 345).

4. Set workgroup settings. Specify a location in Amazon S3 for query results and encryption settings, if needed. You can enforce workgroup settings. For more information, see workgroup settings (p. 428).

   Important
   If you override client-side settings (p. 429), 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. 444) and Creating an Amazon SNS Topic in the Amazon Simple Notification Service Getting Started 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. 435).

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. 424).

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. 424).

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. 424).

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 **Workgroup IAM Policies** (p. 423).

- **Example Policy for Full Access to All Workgroups** (p. 425)
- **Example Policy for Full Access to a Specified Workgroup** (p. 425)
- **Example Policy for Running Queries in a Specified Workgroup** (p. 426)
- **Example Policy for Running Queries in the Primary Workgroup** (p. 427)
- **Example Policy for Management Operations on a Specified Workgroup** (p. 427)
- **Example Policy for Listing Workgroups** (p. 427)
- **Example Policy for Running and Stopping Queries in a Specific Workgroup** (p. 428)
IAM Policies for Accessing Workgroups

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",
            "athena:GetQueryResultsStream",
            "athena:CreateNamedQuery",
            "athena:GetNamedQuery",
            "athena:BatchGetNamedQuery",
            "athena:ListNamedQueries",
            "athena:DeleteNamedQuery",
            "athena:CreatePreparedStatement",
            "athena:GetPreparedStatement",
            "athena:ListPreparedStatements"
         ]
      }
   ]
}
```
IAM Policies for Accessing Workgroups

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
```

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
```
Example 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. 426), as in the following example.

```
arn:aws:athena:us-east-1:123456789012:workgroup/primary
```

Example 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:ListEngineVersions"],
         "Resource": "*"
      },
      {
         "Effect": "Allow",
         "Resource": ["arn:aws:athena:us-east-1:123456789012:workgroup/test_workgroup"]
      }
   ]
}
```

Example Example Policy for Listing Workgroups

The following policy allows all users to list all workgroups:

```
{
   "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:

```
{
   "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:

```
{
   "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 on creating an Amazon S3 bucket, see [Create a Bucket](#).
  • **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. 429). 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. 433).

You can also set query limits (p. 437) 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 clients-side settings for query results location and encryption. 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 and encryption. 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, the notification dialog box displays, as shown in the following example. It 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 and encryption. To check which settings are used for the workgroup, view workgroup’s details (p. 433).

Managing Workgroups

In the 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>Create a Workgroup (p. 430)</td>
<td>Create a new workgroup.</td>
</tr>
<tr>
<td>Edit a Workgroup (p. 432)</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>View the Workgroup's Details (p. 433)</td>
<td>View the workgroup's details, such as its name, description, data usage limits, location of query results, and encryption. You can also verify whether this workgroup enforces its settings, if Override client-side settings is checked.</td>
</tr>
<tr>
<td>Delete a Workgroup (p. 433)</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>Switch between Workgroups (p. 434)</td>
<td>Switch between workgroups to which you have access.</td>
</tr>
<tr>
<td>Copy a Saved Query between Workgroups (p. 434)</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>
<tr>
<td>Enable and Disable a Workgroup (p. 435)</td>
<td>Enable or disable a workgroup. When a workgroup is disabled, its users cannot run queries, or create new named queries. If you have access to it, you can still view metrics, data usage limit controls, workgroup's settings, query history, and saved queries.</td>
</tr>
<tr>
<td>Specify a Workgroup in Which to Run Queries (p. 435)</td>
<td>Before you can run queries, you must specify to Athena which workgroup to use. You must have permissions to the workgroup.</td>
</tr>
</tbody>
</table>

Create a Workgroup

Creating a workgroup requires permissions to CreateWorkgroup API actions. See Access to Athena Workgroups (p. 345) and IAM Policies for Accessing Workgroups (p. 423). If you are adding tags, you also need to add permissions to TagResource. See Tag Policy Examples for Workgroups (p. 454).

To create a workgroup in the console

1. In the Athena console, choose the Workgroup:<workgroup_name> tab. A Workgroups panel displays.
2. In the **Workgroups** panel, choose **Create workgroup**.

3. In the **Create workgroup** dialog box, fill in the fields as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Workgroup name</strong></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><strong>Description</strong></td>
<td>Optional. Enter a description for your workgroup. It can contain up to 1024 characters.</td>
</tr>
</tbody>
</table>
| **Query result location** | Optional. Enter a path to an Amazon S3 bucket or prefix. This bucket and prefix must exist before you 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 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**. |
| **Encrypt query results** | Optional. Encrypt results stored in Amazon S3. If selected, all queries in the workgroup are encrypted.  
  If selected, you can select the **Encryption type**, the **Encryption key** and enter the **KMS Key ARN**.  
  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**. |
| **Update query engine** | 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. 459). |
| **Metrics**            | This field is selected by default. Publish query metrics to CloudWatch. See **Viewing Query Metrics** (p. 438). |
| **Override client-side settings** | 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. 429). |
| **Tags**               | 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, |
### Field Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>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. 448).</td>
<td></td>
</tr>
<tr>
<td>Requester Pays S3 buckets</td>
<td>Optional. Choose <a href="#">Enable queries on Requester Pays buckets in Amazon S3</a> 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 <a href="#">Requester Pays Buckets</a> in the Amazon Simple Storage Service Developer Guide.</td>
</tr>
</tbody>
</table>

4. Choose **Create workgroup**. The workgroup appears in the list in the **Workgroups** panel.

Alternatively, use the API operations to create a workgroup.

**Important**

After you create workgroups, create IAM Policies for Workgroups (p. 423) IAM that allow you to run workgroup-related actions.

### Edit a Workgroup

Editing a workgroup requires permissions to [UpdateWorkgroup](#) API operations. See Access to Athena Workgroups (p. 345) and IAM Policies for Accessing Workgroups (p. 423). If you are adding or editing tags, you also need to have permissions to [TagResource](#). See Tag Policy Examples for Workgroups (p. 454).

**To edit a workgroup in the console**

1. In the Athena console, choose the **Workgroup:<workgroup_name>** tab. A **Workgroups** panel displays, listing all of the workgroups in the account.

2. In the **Workgroups** panel, choose the workgroup that you want to edit. The View details panel for the workgroup displays, with the Overview tab selected.

3. Choose **Edit workgroup**.
4. Change the fields as needed. For the list of fields, see Create workgroup (p. 430). 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. The updated workgroup appears in the list in the Workgroups panel.

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 and encryption configuration. If a workgroup has data usage limits, they are also displayed.

To view the workgroup's details

• In the Workgroups panel, choose the workgroup that you want to edit. The View details panel for the workgroup displays, with the Overview tab selected. The workgroup details display, as in the following example:

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. 427).

To delete a workgroup in the console

1. In the Athena console, choose the Workgroup:<workgroup_name> tab. A Workgroups panel displays.
2. In the Workgroups panel, choose the workgroup that you want to delete. The View details panel for the workgroup displays, with the Overview tab selected.
3. Choose Delete workgroup, and confirm the deletion.

To delete a workgroup with the API operation, use the DeleteWorkGroup action.

Switch between 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 between workgroups

1. In the Athena console, choose the Workgroup:<workgroup_name> tab. A Workgroups panel displays.
2. In the Workgroups panel, choose the workgroup that you want to switch to, and then choose Switch workgroup.
3. Choose Switch. The console shows the Workgroup: <workgroup_name> tab with 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 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 saved query that you want to copy. If prompted Are you sure you want to open this query?, choose Open query. 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. 434) to the destination workgroup, or create a workgroup (p. 430), 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, choose the **Workgroup:<workgroup_name>** tab. A **Workgroups** panel displays.
2. In the **Workgroups** panel, choose the workgroup, and then choose **Enable workgroup** or **Disable workgroup**. 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

Before you can run queries, you must specify to Athena which workgroup to use. You need to have permissions to the workgroup.

**To specify a workgroup to Athena**

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. 423).
2. To specify the workgroup to Athena, use one of these options:
   - If you are accessing Athena via the console, set the workgroup by switching workgroups (p. 434).
   - 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:

```java
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:

```java
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. 87).

### 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. 423).
- 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. 435).
- 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. 435).

- If the workgroup in which a query will run is configured with an enforced query results location (p. 429), 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: <code>arn:aws:iam::123456789012:user/abc</code> is not authorized to perform: athena:StartQueryExecution on resource: <code>arn:aws:athena:us-east-1:123456789012:workgroup/workgroupname</code></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 <code>OutputLocation</code> (client-side), or in the workgroup, using <code>WorkGroupConfiguration</code>.</td>
</tr>
<tr>
<td>The Create Table As Select query failed because it was submitted with an 'external_location' property to an Athena Workgroup that enforces a centralized output location for all queries. Please remove the 'external_location' property and resubmit the query.</td>
<td>If the workgroup in which a query runs is configured with an enforced query results location (p. 429), and you specify an external_location for the CTAS query. In this case, remove the external_location and rerun the query.</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. 438). 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. 438)
- Monitoring Athena Queries with CloudWatch Metrics (p. 438)
- Monitoring Athena Queries with CloudWatch Events (p. 442)
- Setting Data Usage Control Limits (p. 444)

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. Choose the Workgroup tab.
3. Choose the workgroup that you want to modify, and then choose View details.
4. Choose Edit workgroup.
5. On the Edit workgroup page, under Metrics, select or clear the Publish query metrics to AWS CloudWatch option.

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 Publish to CloudWatch 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 CANCELLED
- QueryType – DML or DDL
- WorkGroup – name of the workgroup

For more information, see the List of CloudWatch Metrics and Dimensions for Athena (p. 441) later in this topic.

To view query metrics for a workgroup in the console

2. Choose the Workgroup:<name> tab.
   
   To view a workgroup’s metrics, you don’t need to switch to it and can remain in another workgroup. You do need to select the workgroup from the list. You also must have permissions to view its metrics.
3. Select the workgroup from the list, and then choose View details. If you have permissions, the workgroup’s details display in the Overview tab.
4. 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. 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 down arrow next to the refresh icon to choose the Auto refresh option and a refresh interval for the metrics display.

**To view metrics in the Amazon CloudWatch console**

2. In the navigation pane, choose 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 amount of data in megabytes 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 or CTAS 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>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. Valid statistics: QUEUED, RUNNING, SUCCEEDED, FAILED, or CANCELLED.</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>
</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.

```json
{
    "source": ["aws.athena"],
    "detail-type": ["Athena Query State Change"],
    "detail": {
        "currentState": ["SUCCEEDED"]
    }
}
```

Athena Query State Change Event

The following is the format of an Athena Query State Change event.

```json
{
    "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",
    "detail": {
        "currentState": ["SUCCEEDED"]
    }
}
```
Output Properties

The JSON output includes the following properties.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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"
}
```
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. 444).
- **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 to notify an administrator when the limit is breached. For detailed steps, see To create a per-workgroup data usage control (p. 446). 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. 435).

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](https://docs.aws.amazon.com/AmazonS3/latest/userguide/lifecycle-configuration.html) in the *Amazon Simple Storage Service Developer 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. Choose the **Workgroup** tab.

   To create a data usage control for a query in a particular workgroup, you don't need to switch to it and can remain in another workgroup. You do need to select the workgroup from the list and have permissions to edit the workgroup.

3. Select the workgroup from the list, and then choose **View details**. If you have permissions, the workgroup's details display in the **Overview** tab.

4. Choose the **Data usage controls** tab.

5. In the **Per query data usage control** section, specify the field values, as follows:
   - For **Data limits**, specify a value between 10000 KB (minimum) and 7000000 TB (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 (KB, MB, GB, or TB).
- The default **Action** is to cancel the query if it exceeds the limit. This setting cannot be changed.

6. Choose **Create** if you are creating a new limit, or **Update** if you are editing an existing limit. If you are editing an existing limit, refresh the **Overview** tab to see the updated limit.

To create a per-workgroup data usage control

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 control limits per workgroup. If the limit is exceeded, you can choose to take action, such as send an Amazon SNS alarm notification to the specified users.

2. Choose the **Workgroup** tab.

   To create a data usage control for a particular workgroup, you don’t need to switch to it and can remain in another workgroup. You do need to select the workgroup from the list and have permissions to edit the workgroup.

3. Select the workgroup from the list, and then choose **View details**. If you have edit permissions, the workgroup’s details display in the **Overview** tab.
4. Choose the **Data usage controls** tab, and scroll down. Then choose **Workgroup data usage controls** to create a new limit or edit an existing limit. The **Create workgroup data usage control** dialog displays.

5. Specify field values as follows:

   - For **Data limits**, specify a value between 10 MB (minimum) and 7000000 TB (maximum).
Setting Data Usage Control Limits

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 time period, choose a time period from the drop-down list.
- For Action, choose an Amazon SNS topic from the drop-down list, if you have one configured. Or, choose Create an Amazon 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 Creating an Amazon SNS Topic in the Amazon Simple Notification Service Getting Started Guide.

6. Choose Create if you are creating a new limit, or Save if you are editing an existing limit. If you are editing an existing limit, refresh the Overview tab for the workgroup to see the updated limit.
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. 453).

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. 448).
• 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. 449)
• Adding and Deleting Tags on an Individual Workgroup (p. 449)

Displaying Tags for Individual Workgroups

You can display tags for an individual workgroup in the Athena console.

To view a list of tags for a workgroup, select the workgroup, choose View Details, and then choose the Tags tab. The list of tags for the workgroup displays. You can also view tags on a workgroup if you choose Edit Workgroup.

To search for tags, choose the Tags tab, and then enter a tag name into the search tool.

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 creating a new workgroup

2. On the navigation menu, choose the Workgroups tab.
3. Choose Create workgroup and fill in the values, as needed. For detailed steps, see Create a Workgroup (p. 430).
4. 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. 448).
5. When you are done, choose Create Workgroup.

To add or edit a tag to an existing workgroup

1. Open the Athena console at https://console.aws.amazon.com/athena/, choose the Workgroups tab, and select the workgroup.
2. Choose View details.
3. Do one of the following:
   - Choose the Tags tab, and then choose Manage tags.
   - Choose Edit workgroup, and then scroll down to the Tags section.
4. Specify the key and value for each tag. For more information, see Tag Restrictions (p. 448).
5. Choose Save.

To delete a tag from an individual workgroup

1. Open the Athena console, and then choose the Workgroups tab.
2. In the workgroup list, select the workgroup, and then choose View details.
3. Do one of the following:
   - Choose the Tags tab, and then choose Manage tags.
   - Choose Edit workgroup, and then scroll down to the Tags section.
4. In the list of tags, select the delete button (x) 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.

<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>
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
List<String> tagKeys = new ArrayList<>();
tagKeys.add("tagKey2");
UntagResourceRequest request = new UntagResourceRequest()
    .withTagKeys(tagKeys);
client.untagResource(request);
```

The following example removes `tagKey2` from the data catalog `datacatalogA`:

```java
List<String> tagKeys = new ArrayList<>();
```
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**

```
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.

```
  Key=Color,Value=Orange,Key=Time,Value=Now
```

To show the result, use the `list-tags-for-resource` command.

For information on adding tags when using the `create-data-catalog` command, see Registering a Catalog: create-data-catalog (p. 62).
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:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": [
        "athena:ListWorkGroups",
        "athena:ListEngineVersions",
        "athena:ListDataCatalogs",
        "athena:ListDatabases",
        "athena:GetDatabase",
        "athena:ListTableMetadata",
        "athena:GetTableMetadata"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Deny",
      "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"
      ],
      "Resource": "arn:aws:athena:us-east-1:123456789012:workgroup/workgroupB" // Replace with your specific resource ARN
    }
  ]
}
```
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.
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": ["arn:aws:athena:us-east-1:123456789012:workgroup/*"]
      },
      {
         "Effect": "Allow",
         "Action": [
            "athena:CreateDataCatalog",
            "athena:GetCatalog",
            "athena:UpdateDataCatalog",
            "athena:DeleteDataCatalog",
            "athena:ListDatabases",
            "athena:GetDatabase",
            "athena:GetTableMetadata"
         ],
         "Resource": "*"
      }
   ]
}
```
Tag Policy Examples for Data Catalogs

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.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Deny",
         "Action": [
            "athena:CreateDataCatalog",
            "athena:GetDataCatalog",
            "athena:UpdateDataCatalog",
            "athena:DeleteDataCatalog",
            "athena:GetDatabase",
            "athena:ListDatabases",
            "athena:GetTableMetadata",
            "athena:ListTableMetadata",
            "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.

```
{
   "Version": "2012-10-17",
   "Statement": [
```json
{
   "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 Personal Health Dashboard. Your AWS Personal Health Dashboard notifies you about events that can affect your AWS services or account. For more information about AWS Personal Health Dashboard, see Getting started with the AWS Personal Health Dashboard.

Engine versioning is configured per workgroup (p. 420). 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. 463).

Topics
- Changing Athena Engine Versions (p. 459)
- Athena Engine Version Reference (p. 463)

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. 459)
- Changing the Engine Version (p. 460)
- Specifying the Engine Version When You Create a Workgroup (p. 462)
- Testing Queries in Advance of an Engine Version Upgrade (p. 463)
- Troubleshooting Queries That Fail (p. 463)

Finding the Query Engine Version for a Workgroup

The Athena console shows the engine version for the current workgroup at the bottom of the Query Editor window, as in the following example:
You can also use the Workgroups page to find the current engine version for any workgroup.

To find the current engine version for any workgroup

1. In the Athena console, choose the Workgroup tab.
2. In the list of workgroups, find the workgroup that you want.

The engine version is shown in the Query engine version column for the workgroup, as in the following example.

Changing the Engine Version

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.

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. 424).

To let Athena decide when to upgrade the workgroup

1. In the Athena console, choose the Workgroup tab.
2. In the list of workgroups, choose the workgroup that you want to configure.
3. Choose **View details**.
4. Choose **Edit workgroup**.
5. Under **Query engine version**, for **Update query engine**, choose **Let Athena choose when to upgrade your workgroup**. This is the default setting.
6. Choose **Save**.

The workgroup's **Query engine update status** is set to **Pending automatic upgrade**. When the update occurs, Athena will notify you in the Athena console and on your **AWS Personal Health Dashboard**. The workgroup continues to use the current engine version until the update.

**To manually choose an engine version**

1. In the Athena console, choose the **Workgroup** tab.
2. In the list of workgroups, choose the workgroup that you want to configure.
3. Choose **View details**.
4. Choose **Edit workgroup**.
5. Under **Query engine version**, for **Update query engine**, choose **Manually choose an engine version now**.
6. Choose the engine version that you want the workgroup to use, and then choose **Save**.
7. If you chose an older engine version, a **Confirm Athena engine version** warning appears. Read the warning, and then do one of the following:

   - To cancel the engine version change, choose **Cancel**.
   - To confirm the engine version change, enter **confirm** in the box, and then choose **Confirm**.

The **Query engine update status** for the workgroup shows **Manually set**.
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. 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. 424).

To specify the engine version when you create a workgroup

1. In the Athena console, choose the Workgroup tab.
2. On the Workgroups page, choose Create workgroup.
3. On the Create workgroup page, under the Query engine version, for Update query engine, do one of the following:
   - Choose Let Athena choose when to upgrade your workgroup. This is the default setting.
   - Choose Manually choose an engine version now, and then choose an engine version.
4. Enter information for the other fields as necessary. For information about the other fields, see Create a Workgroup (p. 430).
5. 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 may 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 in the Athena Query Editor and on the Workgroups page. For more information, see Finding the Query Engine Version for a Workgroup (p. 459).
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. 462).
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. 463) 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 visit the Amazon Athena Forum.

Troubleshooting Queries That Fail

If a query fails after an engine version upgrade, use the Athena Engine Version Reference (p. 463) 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 visit the Amazon Athena Forum.

Athena Engine Version Reference

This section lists the changes to the Athena query engine since Athena engine version 1. Athena engine version 1 was Athena's initial engine version and was based on Presto 0.172.

Athena engine version 2

Athena engine version 2, which is based on Presto 0.217, 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. 464)
- Grouping, Join, and Subquery Improvements (p. 464)
- Datatype Enhancements (p. 464)
- Added Functions (p. 465)
- Performance Improvements (p. 466)
• JSON-Related Improvements (p. 468)
• Breaking Changes (p. 469)
  • Bug Fixes (p. 469)
  • Changes to Geospatial Functions (p. 469)
  • ANSI SQL Compliance (p. 470)
  • Replaced Functions (p. 470)
  • Limits (p. 471)

Improvements and New Features

• Federated queries – Federated queries are supported in Athena engine version 2. For more information, see Using Amazon Athena Federated Query (p. 68).
• Geospatial functions – More than 25 geospatial functions have been added. For more information, see New Geospatial Functions in Athena engine version 2 (p. 238).
• Nested schema – Support has been added for reading nested schema, which reduces cost.
• 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.

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.

### 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 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. 238).

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.

**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

- **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.
- **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.

• **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.

**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)`).

**Subquery Performance**

• **Correlated EXITS 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_Within()`, `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** – `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.
- **round(x, d) and truncate(x, d)** – The parameter `d` in the signature of functions `round(x, d)` and `truncate(x, d)` is now of type `INTEGER`. Previously, `d` could be of type `BIGINT`.
- **map() with duplicate keys** – `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** – `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 `corr()`, `covar_samp()`, `regr_intercept()`, and `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_TOSECOND and INTERVAL_YEAR_TO_MONTH datatypes** – You can no longer use `SUM(NULL)` directly. In order to use `SUM(NULL)`, cast `NULL` to a data type like `BIGINT`, `DECIMAL`, `REAL`, `DOUBLE`, `INTERVAL_DAY_TOSECOND` 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. 237).
- **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 `ST_area()` to calculate the area for `VARBINARY` input in Well-Known Binary (WKB) format, pass the input to `ST_GeomFromBinary()` first, for example:

```
ST_area(ST_GeomFromBinary(<wkb_varbinary_value>))
```

To use `ST_area()` to calculate the area for `VARBINARY` input in legacy binary format, pass the same input to the `ST_GeomFromLegacyBinary()` function first, for example:

```
ST_area(ST_GeomFromLegacyBinary(<legacy_varbinary_value>))
```

`ST_ExteriorRing()` and `ST_Polygon()` – `ST_ExteriorRing()` (p. 230) and `ST_Polygon()` (p. 227) now accept only polygons as inputs. Previously, these functions erroneously accepted other geometries.

`ST_Distance()` – As required by the SQL/MM specification, the `ST_Distance()` (p. 232) 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. For equivalent syntax, use `SELECT * FROM "<table_name>$partitions" or SHOW PARTITIONS`. For more information, see Listing Partitions for a Specific Table (p. 305).
• **Replaced geospatial functions** – For a list of geospatial functions whose names have changed, see
  Geospatial Function Name Changes in Athena engine version 2 (p. 237).

**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: $\text{min}(\text{col}, n), \text{max}(\text{col}, n), \text{min}_\text{by}(\text{col1, col2, n}), \text{and max}_\text{by}(\text{col1, col2, n}).$

• **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.
Data Types in Amazon Athena

When you run CREATE TABLE (p. 511), 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 combines two different implementations of the integer data type, as follows:
  - **INT** – In Data Definition Language (DDL) queries, Athena uses the INT data type.
  - **INTEGER** – In DML queries, Athena uses 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 double-precision floating point number.
- **FLOAT** – A 32-bit single-precision floating point number. 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. 582) 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).
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.

  ```sql
  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. 140).

- **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. 140).

- **ARRAY**
  ```sql
  <data_type>
  ```

- **MAP**
  ```sql
  <primitive_type, data_type>
  ```

- **STRUCT**
  ```sql
  <col_name : data_type [COMMENT col_comment] , ...>
  ```

### DML Queries, Functions, and Operators

Athena DML query statements are based on Presto 0.172 for Athena engine version 1 and Presto 0.217 for Athena engine version 2. For information about Athena engine versions, see Athena Engine Versioning (p. 459).

For links to subsections of the Presto function documentation, see Presto Functions (p. 492).

Athena does not support all of Presto’s features, and there are some significant differences. For more information, see the topics for specific statements in this section and Considerations and Limitations (p. 528).

For information about DDL statements, see DDL Statements (p. 502). For a list of unsupported DDL statements, see Unsupported DDL (p. 503).

**Topics**

- SELECT (p. 474)
- INSERT INTO (p. 479)
- UNLOAD (p. 482)
- Using the EXPLAIN Statement in Athena (p. 484)
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. 528) and Running SQL Queries Using Amazon Athena (p. 162). For help getting started with querying data in Athena, see Getting Started (p. 8).

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] [, ... ]
[ 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. 107).

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:

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.
SELECT

[ ALL | DISTINCT ] select_expr

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'
SELECT • [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.

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_mutable/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?](https://docs.aws.amazon.com/athena/latest/ug/how-can-i-see-aws-s3-source-file-for-row-in-an-athena-table.html).

### 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**

O'Reilly

### 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><a href="https://docs.aws.amazon.com/athena/latest/ug/sql-ref-syntax-select.html">Running SQL Queries Using Amazon Athena</a> (p. 162)</td>
</tr>
<tr>
<td>Using <code>SELECT</code> to create a table</td>
<td>[Creating a Table from Query Results (CTAS)] (p. 177)</td>
</tr>
<tr>
<td>Inserting data from a <code>SELECT</code> query into another table</td>
<td><a href="https://docs.aws.amazon.com/athena/latest/ug/sql-ref-syntax-insert.html">INSERT INTO</a> (p. 479)</td>
</tr>
<tr>
<td>Using built-in functions in <code>SELECT</code> statements</td>
<td><a href="https://docs.aws.amazon.com/athena/latest/ug/presto-functions.html">Presto Functions in Amazon Athena</a> (p. 492)</td>
</tr>
</tbody>
</table>
For Information About This | See This
---|---
Using user defined functions in `SELECT` statements | Querying with User Defined Functions (p. 262)
Querying Data Catalog metadata | Querying AWS Glue Data Catalog (p. 303)

**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.

**Important**

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. 320). 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.

**Supported Formats and SerDes**

You can run an `INSERT` query on tables created from data with the following formats and SerDes.

<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. 182).

**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. 192).

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. 192).
- For inserting unpartitioned data into a partitioned table, see Using CTAS and INSERT INTO for ETL and Data Analysis (p. 186).

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. 164).

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. 164) 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. 474).

**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:

```sql
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:

```sql
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:

```sql
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, Output Files, and Query History (p. 162).

**Synopsis**

```sql
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:

```sql
INSERT INTO cities
VALUES (1,'Lansing','MI','Si quaeris peninsulam amoenam circumspice')
```

Insert two rows into the cities table:
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:

- **Engine version** – Athena engine version 2 is required.
- **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. 164).
- **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 Querying with Prepared Statements (p. 195).
- **Service quotas** – `UNLOAD` uses DML query quotas. For quota information, see Service Quotas (p. 558).

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.

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.
Using the EXPLAIN Statement 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.

Considerations and Limitations

The EXPLAIN statement in Athena has 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**.
- Athena does not support EXPLAIN ANALYZE, which collects runtime statistics.

Syntax – Athena engine version 1

```
EXPLAIN [ ( option [, ...]) ] statement
```

*option* can be one of the following:

```
FORMAT { TEXT | GRAPHVIZ }
TYPE { LOGICAL | DISTRIBUTED | VALIDATE }
```

Syntax – Athena engine version 2

```
EXPLAIN [ ( option [, ...]) ] statement
```

*option* can be one of the following:

```
FORMAT { TEXT | GRAPHVIZ | JSON }
TYPE { LOGICAL | DISTRIBUTED | VALIDATE | IO }
```

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.

Examples

The following examples progress from the more straightforward to the more complex. Example results are in text format.

Example 1. Use the EXPLAIN statement to show a text query plan

```sql
EXPLAIN
SELECT
    request_timestamp,
    elb_name,
FROM
    table1;
```
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

Example 2. Use the EXPLAIN statement to graph a query plan

EXPLAIN (FORMAT GRAPHVIZ)
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 = 5566684

Results

Query Plan
digraph logical_plan {
  subgraph cluster_graphviz_plan {
    plannode_1[label="{Output[c_custkey, o_orderkey, o_orderstatus]}", style="rounded, filled",
    shape=record, fillcolor=white];
    plannode_2[label="{ExchangeNode[GATHER]|"c_custkey", \"o_orderstatus\", \"o_orderkey\"}",
    style="rounded, filled", shape=record, fillcolor=gold];
    plannode_3[label="{InnerJoin}", style="rounded, filled", shape=record, fillcolor=orange];
    plannode_4[label="{Filter|("c_custkey" = 5566684)}", style="rounded, filled",
    shape=record, fillcolor=yellow];
    plannode_5[label="{TableScan[awsdatacatalog:HiveTableHandle{schemaName=tpch100, tableName=customer, analyzePartitionValues=Optional.empty}]}",
    style="rounded, filled", shape=record, fillcolor=deepskyblue];
    plannode_6[label="{ExchangeNode[GATHER]|\"o_orderstatus\", \"o_orderkey\"}",
    style="rounded, filled", shape=record, fillcolor=gold];
    plannode_7[label="{ExchangeNode[REPLICATE]|\"o_orderstatus\", \"o_orderkey\"}",
    style="rounded, filled", shape=record, fillcolor=gold];
    plannode_8[label="{Project}", style="rounded, filled", shape=record, fillcolor=bisque];
    plannode_9[label="{Filter|("o_custkey" = 5566684)}", style="rounded, filled",
    shape=record, fillcolor=yellow];
    plannode_10[label="{TableScan[awsdatacatalog:HiveTableHandle{schemaName=tpch100, tableName=orders, analyzePartitionValues=Optional.empty}]}",
    style="rounded, filled", shape=record, fillcolor=deepskyblue];
  }
plannode_1, plannode_2;
plannode_2, plannode_3;
plannode_3, plannode_4;
plannode_4, plannode_5;
plannode_3, plannode_6;
plannode_6, plannode_7;
plannode_7, plannode_8;
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`.

```sql
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)
```
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=2015
o_orderdate=1998
o_orderdate=1995
o_orderdate=1993
o_orderdate=1997
o_orderdate=1992
o_orderdate=1996
```

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]]
  - RemoteExchange[GATHER] => [[o_orderkey, o_custkey, o_orderdate]]
    - TableScan[awsdatalake:HiveTableHandle{schemaName=tpch100,
      tableName=orders_partitioned,
      analyzePartitionValues=Optional.empty}] => [[o_orderkey, o_custkey, o_orderdate]]
      LAYOUT: tpch100.orders_partitioned
        o_orderdate := o_orderdate:string:-1:PARTITION_KEY
        o_custkey := o_custkey:int:1:REGULAR
        o_orderkey := o_orderkey:int:0:REGULAR
```

The bold text in the result shows that the predicate `o_orderdate = '1995'` was applied on the `PARTITION_KEY`.

**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.

```
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
```

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 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.
**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**

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. 490).

**Additional Resources**

For additional information about EXPLAIN queries, see the following resources.

- Presto 0.172 EXPLAIN documentation
- Presto 0.217 EXPLAIN documentation
- Explain the EXPLAIN video on YouTube (20:18)
Understanding Athena EXPLAIN Statement Results

This topic provides a brief guide to the operational terms used in Athena EXPLAIN statement results.

**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).

- **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 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 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 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.

### Presto Functions in Amazon Athena

Athena supports some, but not all, of Presto's functions and features. For information, see [Considerations and Limitations](#). For a list of the time zones that can be used with the AT TIME ZONE operator, see [Supported Time Zones](#).

### Athena engine version 2

Athena engine version 2 is based on [Presto 0.217](#). For information about related functions, operators, and expressions, see [Presto 0.217 Functions and Operators](#) and the following specific sections from the Presto documentation. For the geospatial functions in Athena engine version 2, see [Geospatial Functions in Athena engine version 2](#).

- 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
Athena engine version 1

Athena engine version 1 is based on Presto 0.172. For information about related functions, operators, and expressions, see Presto 0.172 Functions and Operators and the following specific sections from the Presto documentation. For the geospatial functions in Athena engine version 1, see Geospatial Functions in Athena engine version 1 (p. 239).

- 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.

- Africa/Abidjan
- Africa/Accra
- Africa/Addis_Ababa
- Africa/Algiers
- Africa/Asmara
Supported Time Zones

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-Novo
Africa/Sao_Tome
Africa/Timbuktu
Africa/Tripoli
Africa/Tunis
Africa/Windhoek
America/Adak
America/Anchorage
America/Anguilla
America/Antigua
America/Araguaina
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
Supported Time Zones

- 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/Port_Wayne
- America/Porto_Alegre
- America/Glace_Bay
- America/Godthab
- 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
Supported Time Zones

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/Miguelon
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
America/Porto_Acre
America/Porto_Velho
America/Puerto_Rico
America/Punta_Arenas
America/Rainy_River
America/Rankin_Inlet
America/Recife
America/Regina
America/Resolute
America/Rio_Branco
America/Rosario
America/Santa_Isabel
America/Santarem
America/Santiago
America/Santo_Domingo
America/Sao_Paulo
America/Scoresbysund
America/Shiprock
America/Sitka
America/St_Barthelemy
America/St_Johns
<table>
<thead>
<tr>
<th>Time Zone</th>
</tr>
</thead>
<tbody>
<tr>
<td>America/St_Kitts</td>
</tr>
<tr>
<td>America/St_Lucia</td>
</tr>
<tr>
<td>America/St_Thomas</td>
</tr>
<tr>
<td>America/St_Vincent</td>
</tr>
<tr>
<td>America/Swift_Current</td>
</tr>
<tr>
<td>America/Tegucigalpa</td>
</tr>
<tr>
<td>America/Thule</td>
</tr>
<tr>
<td>America/Thunder_Bay</td>
</tr>
<tr>
<td>America/Tijuana</td>
</tr>
<tr>
<td>America/Toronto</td>
</tr>
<tr>
<td>America/Tortola</td>
</tr>
<tr>
<td>America/Vancouver</td>
</tr>
<tr>
<td>America/Virgin</td>
</tr>
<tr>
<td>America/Whitehorse</td>
</tr>
<tr>
<td>America/Winnipeg</td>
</tr>
<tr>
<td>America/Yakutat</td>
</tr>
<tr>
<td>America/Yellowknife</td>
</tr>
<tr>
<td>Antarctica/Casey</td>
</tr>
<tr>
<td>Antarctica/Devis</td>
</tr>
<tr>
<td>Antarctica/DumontDUrville</td>
</tr>
<tr>
<td>Antarctica/Macquarie</td>
</tr>
<tr>
<td>Antarctica/Mawson</td>
</tr>
<tr>
<td>Antarctica/McMurdo</td>
</tr>
<tr>
<td>Antarctica/Palmer</td>
</tr>
<tr>
<td>Antarctica/Rothera</td>
</tr>
<tr>
<td>Antarctica/South_Pole</td>
</tr>
<tr>
<td>Antarctica/Syowa</td>
</tr>
<tr>
<td>Antarctica/Troll</td>
</tr>
<tr>
<td>Antarctica/Vostok</td>
</tr>
<tr>
<td>Arctic/Longyearbyen</td>
</tr>
<tr>
<td>Asia/Aden</td>
</tr>
<tr>
<td>Asia/Almaty</td>
</tr>
<tr>
<td>Asia/Amman</td>
</tr>
<tr>
<td>Asia/Anadyr</td>
</tr>
<tr>
<td>Asia/Aqtobe</td>
</tr>
<tr>
<td>Asia/Aqtau</td>
</tr>
<tr>
<td>Asia/Ashgabat</td>
</tr>
<tr>
<td>Asia/AshKhabad</td>
</tr>
<tr>
<td>Asia/Atyrau</td>
</tr>
<tr>
<td>Asia/Baghdad</td>
</tr>
<tr>
<td>Asia/Bahrain</td>
</tr>
<tr>
<td>Asia/Baku</td>
</tr>
<tr>
<td>Asia/Bangkok</td>
</tr>
<tr>
<td>Asia/Barnaul</td>
</tr>
<tr>
<td>Asia/Beirut</td>
</tr>
<tr>
<td>Asia/Bishkek</td>
</tr>
<tr>
<td>Asia/Brunei</td>
</tr>
<tr>
<td>Asia/Calcutta</td>
</tr>
<tr>
<td>Asia/Chita</td>
</tr>
<tr>
<td>Asia/Choibalsan</td>
</tr>
<tr>
<td>Asia/Chongqing</td>
</tr>
<tr>
<td>Asia/Chungking</td>
</tr>
<tr>
<td>Asia/Colombo</td>
</tr>
<tr>
<td>Asia/Dacca</td>
</tr>
<tr>
<td>Asia/Damascus</td>
</tr>
<tr>
<td>Asia/Dhaka</td>
</tr>
<tr>
<td>Asia/Dili</td>
</tr>
<tr>
<td>Asia/Dubai</td>
</tr>
<tr>
<td>Asia/Dushanbe</td>
</tr>
<tr>
<td>Asia/Gaza</td>
</tr>
<tr>
<td>Asia/Harbin</td>
</tr>
<tr>
<td>Asia/Hebron</td>
</tr>
<tr>
<td>Asia/Ho_Chi Minh</td>
</tr>
<tr>
<td>Asia/Hong_Kong</td>
</tr>
<tr>
<td>Asia/Hovd</td>
</tr>
<tr>
<td>Asia/Irkutsk</td>
</tr>
</tbody>
</table>

497
Supported Time Zones

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/Novokuzensk
Asia/Novosibirsk
Asia/Omsk
Asia/Oral
Asia/Phnom_Penh
Asia/Pontianak
Asia/Pyongyang
Asia/Qatar
Asia/Qyzylorda
Asia/Rangoon
Asia/Riyadh
Asia/Saigon
Asia/Sakhalin
Asia/Samarkand
Asia/Seoul
Asia/Shanghai
Asia/Singapore
Asia/Srednekolymsk
Asia/Taipei
Asia/Tashkent
Asia/Tbilisi
Asia/Tehran
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
Supported Time Zones

Amazon Athena User Guide

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
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
Supported Time Zones

- 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
- Europe/Zaporozhye
- Europe/Zurich

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/Apiag
Pacific/Auckland
Pacific/Bougainville
Pacific/Chatham
Pacific/Chuuk
Pacific/Easter
Pacific/Efate
Pacific/Enderbury
Pacific/Fakaofo
Pacific/Fiji
Pacific/Punafuti
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
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
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. 503).

Topics

- Unsupported DDL (p. 503)
- ALTER DATABASE SET DBPROPERTIES (p. 504)
- ALTER TABLE ADD COLUMNS (p. 504)
- ALTER TABLE ADD PARTITION (p. 505)
- ALTER TABLE DROP PARTITION (p. 506)
- ALTER TABLE RENAME PARTITION (p. 507)
- ALTER TABLE REPLACE COLUMNS (p. 507)
- ALTER TABLE SET LOCATION (p. 508)
- ALTER TABLE SET TBLPROPERTIES (p. 508)
- CREATE DATABASE (p. 509)
- CREATE TABLE (p. 511)
- CREATE TABLE AS (p. 514)
- CREATE VIEW (p. 516)
- DESCRIBE TABLE (p. 517)
- DESCRIBE VIEW (p. 518)
- DROP DATABASE (p. 518)
- DROP TABLE (p. 519)
- DROP VIEW (p. 519)
- MSCK REPAIR TABLE (p. 520)
- SHOW COLUMNS (p. 522)
- SHOW CREATE TABLE (p. 522)
- SHOW CREATE VIEW (p. 523)
- SHOW DATABASES (p. 523)
- SHOW PARTITIONS (p. 524)
- SHOW TABLES (p. 526)
- SHOW TBLPROPERTIES (p. 527)
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 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
- LOCK TABLE
- REVOKE ROLE
- ROLLBACK
- SHOW COMPACTIONS
- SHOW CURRENT ROLES
- SHOW GRANT
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

```
ALTER TABLE table_name
[PARTITION
  (partition_col1_name = partition_col1_value
  [,partition_col2_name = partition_col2_value][,...])]
```
ALTER TABLE ADD PARTITION

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 [,...])

Adds columns after existing columns but before partition columns.

Examples

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 (p. 114).

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. 205).

For information about the resource-level permissions required in IAM policies (including glue:CreatePartition), see AWS Glue API Permissions: Actions and Resources Reference and Fine-Grained Access to Databases and Tables in the AWS Glue Data Catalog (p. 335). For troubleshooting information about permissions when using Athena, see the Permissions (p. 535) section of the Troubleshooting in Athena (p. 530) topic.

Synopsis

ALTER TABLE table_name ADD [IF NOT EXISTS] PARTITION (partition_col1_name = partition_col1_value [,...])
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 DROP PARTITION

Drops one or more specified partitions for the named table.

Synopsis

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

ALTER TABLE orders
DROP PARTITION (dt = '2014-05-14', country = 'IN');

ALTER TABLE orders
DROP PARTITION (dt = '2014-05-14', country = 'IN'), PARTITION (dt = '2014-05-15', country = 'IN');

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. 114).

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. 150) 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_col1_name = partition_col1_value
 [, partition_col2_name = partition_col2_value][,...])]  
REPLACE COLUMNS (col_name data_type [, col_name data_type, ...])
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. 109).

**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. 527) command.

Apache Hive Managed tables are not supported, so setting 'EXTERNAL' = 'FALSE' has no effect.
## Synopsis

```sql
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. 511).</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. 511) and Creating Tables Based on Encrypted Datasets in Amazon S3 (p. 321).</td>
</tr>
<tr>
<td>orc.compress</td>
<td>Specifies a compression format for data in ORC format. For more information, see ORC SerDe (p. 155).</td>
</tr>
<tr>
<td>parquet.compression</td>
<td>Specifies a compression format for data in Parquet format. For more information, see Parquet SerDe (p. 157).</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. 119).</td>
</tr>
<tr>
<td>skip.header.line</td>
<td>Ignores headers in data when you define a table. For more information, see Ignoring Headers (p. 150).</td>
</tr>
<tr>
<td>storage.location</td>
<td>Specifies a custom Amazon S3 path template for projected partitions. For more information, see Setting up Partition Projection (p. 121).</td>
</tr>
</tbody>
</table>

### Example

```sql
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.

### Synopsis

```sql
CREATE (DATABASE|SCHEMA) [IF NOT EXISTS] database_name
[COMMENT 'database_comment']
```
CREATE DATABASE

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
  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:

```json
{
  "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.

#### Synopsis

```
CREATE EXTERNAL TABLE [IF NOT EXISTS]
    [db_name.]
    [table_name [((col_name data_type [COMMENT col_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]/']
    [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. All tables created in Athena, except for those created using CTAS (p. 514), must be EXTERNAL. 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.** Athena combines two different implementations of the `INTEGER` data type. In Data Definition Language (DDL) queries, Athena uses the `INT` data type. In all other queries, Athena uses the `INTEGER` data type, 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**

• **FLOAT**

• **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)`.

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 and time types**

  • **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. 140)](#).

  • **TIMESTAMP** Date and time instant in a `java.sql.Timestamp` compatible format up to a maximum resolution of milliseconds, such as `yyyyMMdd 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. 140)](#).

  • **ARRAY** `<data_type>`

  • **MAP** `<primitive_type, data_type>`

  • **STRUCT** `<col_name : data_type [COMMENT col_comment] [, ...]>`

  **[COMMENT table_comment]**

  Creates the `table_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. 114)](#).
Note
After you create a table with partitions, run a subsequent query that consists of the MSCK REPAIR TABLE (p. 520) 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. 505) 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.

[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
• 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. 109). For information about data format and permissions, see Requirements for Tables in Athena and Data in Amazon S3 (p. 100).

Use a trailing slash for your folder or bucket. Do not use file names or glob characters.

Use:
CREATE TABLE AS

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', ] [ 'classification'='aws_glue_classification', ] 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. 318).

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. 29) and Authoring Jobs in Glue in the AWS Glue Developer Guide.

For more information about creating tables, see Creating Tables in Athena (p. 99).

CREATE TABLE AS

Creates a new table populated with the results of a SELECT (p. 474) query. To create an empty table, use CREATE TABLE (p. 511).

For additional information about CREATE TABLE AS beyond the scope of this reference topic, see Creating a Table from Query Results (CTAS) (p. 177).

Topics
- Synopsis (p. 474)
- CTAS Table Properties (p. 515)
- Examples (p. 516)

Synopsis

```
CREATE TABLE table_name
[ WITH ( property_name = expression [, ...] ) ]
AS query
[ WITH [ NO ] DATA ]
```

Where:
CREATE TABLE AS

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. 515).

query

A SELECT (p. 474) 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.

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. 183).

WITH (property_name = expression [, ...], )

external_location = [location]

Optional. The location where Athena saves your CTAS query in Amazon S3, as in the following example:

```
WITH (external_location = 's3://my-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. 429), the query fails with an error message. To see the query results location specified for the workgroup, see the workgroup's details (p. 433).

If your workgroup overrides the client-side setting for query results location, Athena creates your table in the following location:

```
s3://<workgroup-query-results-location>/tables/<query-id>/
```

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. 168) 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>/
```

format = [format]

The data format for the CTAS query results, such as ORC, PARQUET, AVRO, JSON, or TEXTFILE. For example, WITH (format = 'PARQUET'). If omitted, PARQUET is used by default. The name of this parameter, format, must be listed in lowercase, or your CTAS query will fail.
**CREATE VIEW**

```
partitioned_by = ARRAY[ col_name[, ...] ]
```

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.

```
bucketed_by = ARRAY[ bucket_name[, ...] ]
```

An array list of buckets to bucket data. If omitted, Athena does not bucket your data in this query.

```
bucket_count = [int]
```

The number of buckets for bucketing your data. If omitted, Athena does not bucket your data.

```
orc_compression = [format]
```

The compression type to use for ORC data. For example, `WITH (orc_compression = 'ZLIB')`. If omitted, GZIP compression is used by default for ORC and other data storage formats supported by CTAS.

```
parquet_compression = [format]
```

The compression type to use for Parquet data. For example, `WITH (parquet_compression = 'SNAPPY')`. If omitted, GZIP compression is used by default for Parquet and other data storage formats supported by CTAS.

```
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. 183)
- Using CTAS and INSERT INTO for ETL and Data Analysis (p. 186)
- 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. 192)

### 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. 172).

### 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. 175).
Examples

To create a view `test` from the table `orders`, use a query similar to the following:

```sql
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:

```sql
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:

```sql
CREATE OR REPLACE VIEW test AS
SELECT orderkey, orderstatus, totalprice / 4 AS quarter
FROM orders;
```

See also SHOW COLUMNS (p. 522), SHOW CREATE VIEW (p. 523), DESCRIBE VIEW (p. 518), and DROP VIEW (p. 519).

DESCRIBE TABLE

Shows the list of columns, including partition columns, for the named column. This allows you to examine the attributes of a complex column.

Synopsis

```
DESCRIBE [EXTENDED | FORMATTED] [db_name.]table_name [PARTITION partition_spec] [col_name
( [.field_name] | ['.elem$'] | ['.key$'] | ['.value$'])]
```

Parameters

[EXTENDED | FORMATTED]

Determines the format of the output. If you specify EXTENDED, all metadata for the table is output in Thrift serialized form. This is useful primarily for debugging and not for general use. Use FORMATTED or omit the clause to show the metadata in tabular format.

[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;
```

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. 522), SHOW CREATE VIEW (p. 523), SHOW VIEWS (p. 527), and DROP VIEW (p. 519).

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;
```
DROP TABLE

Removes the metadata table definition for the table named `table_name`. When you drop an external table, the underlying data remains intact because all tables in Athena are EXTERNAL.

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 Deleting Views (p. 177).

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. 516), SHOW COLUMNS (p. 522), SHOW CREATE VIEW (p. 523), SHOW VIEWS (p. 527), and DESCRIBE VIEW (p. 518).
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. 506).

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` on the same table until all partitions are added. For more information, see `Partitioning Data` (p. 114).
- For partitions that are not compatible with Hive, use `ALTER TABLE ADD PARTITION` (p. 505) 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.
- Because the command traverses your file system running Amazon S3 `HeadObject` and `GetObject` commands, the cost of bytes scanned can be significant if your file system is large or contains a large amount of data. To estimate costs, see `Amazon S3 pricing` and the `AWS Pricing Calculator`.

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. 331).
- 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. 505).
- **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. 506) instead. Note that `SHOW PARTITIONS` (p. 524) 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 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. 18), 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. 326).

**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:

```
s3://bucket/path/userId=1/
s3://bucket/path/userId=2/
```
SHOW COLUMNS

Lists the names of the columns 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. 306).

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`.`my-table`). 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

SHOW CREATE VIEW

**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: 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](https://aws.amazon.com/cloudformation/). 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](#) and [DROP VIEW](#).

**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. 305) section on the Querying AWS Glue Data Catalog (p. 303) page.
- To view the contents of a partition, see the Query the Data (p. 117) section on the Partitioning Data in Athena (p. 114) 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. 119).
- 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. 506).

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. 150). 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>
</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. 114). The table is partitioned by the `dt` (datetime) column.

```
SHOW PARTITIONS impressions
```

<table>
<thead>
<tr>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dt=2009-04-12-16-00</code></td>
</tr>
<tr>
<td><code>dt=2009-04-13-18-15</code></td>
</tr>
<tr>
<td><code>dt=2009-04-14-00-20</code></td>
</tr>
<tr>
<td><code>dt=2009-04-12-13-00</code></td>
</tr>
<tr>
<td><code>dt=2009-04-13-02-15</code></td>
</tr>
<tr>
<td><code>dt=2009-04-14-12-05</code></td>
</tr>
<tr>
<td><code>dt=2009-04-14-06-15</code></td>
</tr>
<tr>
<td><code>dt=2009-04-12-21-15</code></td>
</tr>
<tr>
<td><code>dt=2009-04-13-22-15</code></td>
</tr>
</tbody>
</table>

Listing Partitions in Sorted Order

To order the partitions in the results list in Athena engine version 2, use the following `SELECT` syntax instead of `SHOW PARTITIONS`.

```
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.

```
SELECT * FROM "flight_delays_csv$partitions" ORDER BY year
```

<table>
<thead>
<tr>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>year</code></td>
</tr>
<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>
</tbody>
</table>

For more information, see the Listing Partitions for a Specific Table (p. 305) section on the Querying AWS Glue Data Catalog (p. 303) 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.

['regular_expression']

Filters the list of tables 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

Example – show all of the tables in the database sampledb

SHOW TABLES IN sampledb

Results

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"

SHOW TABLES IN sampledb '*flights*'　

Results

flights_2016
flights_parquet
view_2016_flights_dfw

Example – show the names of all tables in sampledb that end in the word "logs"

SHOW TABLES IN sampledb '*logs'

Results

alb_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.

**Examples**

```sql
SHOW TBLPROPERTIES orders;
SHOW TBLPROPERTIES orders('comment');
```

**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**

```sql
SHOW VIEWS [IN database_name] LIKE ['regular_expression']
```

**Parameters**

```sql
[IN database_name]
```

Specifies the `database_name` from which views will be listed. If omitted, the database from the current context is assumed.

```sql
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**

```sql
SHOW VIEWS;
```
SHOW VIEWS IN marketing_analytics LIKE 'orders*';

See also SHOW COLUMNS (p. 522), SHOW CREATE VIEW (p. 523), DESCRIBE VIEW (p. 518), and DROP VIEW (p. 519).

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.
- **Parameterized queries** – Parameterized queries are not supported in Athena engine version 1 but are supported in Athena engine version 2. For more information, see Querying with Prepared Statements (p. 195).
- **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. 514). For a workaround, see Using CTAS and INSERT INTO to Create a Table with More Than 100 Partitions (p. 192).
- **Unsupported statements** – The following statements are not supported:
  - CREATE TABLE LIKE is not supported.
  - DESCRIBE INPUT and DESCRIBE OUTPUT is not supported.
  - EXECUTE ... USING is not supported.
  - MERGE statements are not supported.
  - UPDATE statements are not supported.
- **Presto federated connectors** – Presto federated connectors are not supported. Use Amazon Athena Federated Query to connect data sources. For more information, see Using Amazon Athena Federated Query (p. 68).
- **Querying Parquet columns with complex data types** – When you query columns with complex data types (array, map, struct), and are using Parquet for storing data, Athena engine version 1 reads an entire row of data instead of selectively reading only the specified columns. This issue does not occur in Athena engine version 2.
- **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 or S3 Glacier Deep Archive storage classes. Objects in the S3 Glacier storage class 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.
- **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.
Cross-Regional Queries

Athena supports queries across only the following Regions. Queries across other Regions may produce the error message InvalidToken: The provided token is malformed or otherwise invalid.

<table>
<thead>
<tr>
<th>Region Name</th>
<th>Region Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asia Pacific (Tokyo)</td>
<td>ap-northeast-1</td>
</tr>
<tr>
<td>Asia Pacific (Seoul)</td>
<td>ap-northeast-2</td>
</tr>
<tr>
<td>Asia Pacific (Mumbai)</td>
<td>ap-south-1</td>
</tr>
<tr>
<td>Asia Pacific (Singapore)</td>
<td>ap-southeast-1</td>
</tr>
<tr>
<td>Asia Pacific (Sydney)</td>
<td>ap-southeast-2</td>
</tr>
<tr>
<td>Canada (Central)</td>
<td>ca-central-1</td>
</tr>
<tr>
<td>Europe (Frankfurt)</td>
<td>eu-central-1</td>
</tr>
<tr>
<td>Europe (Stockholm)</td>
<td>eu-north-1</td>
</tr>
<tr>
<td>Europe (Ireland)</td>
<td>eu-west-1</td>
</tr>
<tr>
<td>Europe (London)</td>
<td>eu-west-2</td>
</tr>
<tr>
<td>Europe (Paris)</td>
<td>eu-west-3</td>
</tr>
<tr>
<td>South America (São Paulo)</td>
<td>sa-east-1</td>
</tr>
<tr>
<td>US East (N. Virginia)</td>
<td>us-east-1</td>
</tr>
<tr>
<td>US East (Ohio)</td>
<td>us-east-2</td>
</tr>
<tr>
<td>US West (N. California)</td>
<td>us-west-1</td>
</tr>
<tr>
<td>US West (Oregon)</td>
<td>us-west-2</td>
</tr>
</tbody>
</table>
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. 530)
- Data File Issues (p. 530)
- Federated Queries (p. 532)
- JSON Related Errors (p. 532)
- MSCK REPAIR TABLE (p. 533)
- Output Issues (p. 533)
- Partitioning Issues (p. 534)
- Permissions (p. 535)
- Query Syntax Issues (p. 537)
- Throttling Issues (p. 538)
- Views (p. 538)
- Workgroups (p. 538)
- Additional Resources (p. 538)

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.

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 the Glacier storage class

To work around this issue, copy the restored objects back into Amazon S3. For more information, see Considerations and Limitations for SQL Queries in Amazon Athena (p. 528).
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_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_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_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_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. 167) in the Working with Query Results, Output Files, and Query History (p. 162) topic.

Federated Queries

For information on troubleshooting federated queries, see Common_Problems in the awslabs/aws-athena-query-federation section of GitHub.

JSON Related Errors

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. 147) 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. 146). 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.

Multiple JSON records return a SELECT COUNT of 1

If you're using the OpenX JSON SerDe (p. 147), 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. 520) and Troubleshooting (p. 521) sections of the MSCK REPAIR TABLE (p. 520) 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

Currently, Athena outputs files in CSV format only, but you can work around this limitation by using a CTAS query and configuring the `format` table property (p. 515). 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. 167).

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. 506) to remove the stale partitions manually. For more information, see the "Troubleshooting" section of the MSCK REPAIR TABLE (p. 520) 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. 505) 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.<columnName>.interval.unit` (p. 128) 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. 26).

**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 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.

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".

```json
{
  "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 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. 87). 
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 [Presto Functions in Amazon Athena](p. 492) or run the `SHOW FUNCTIONS` statement in the Query Editor. You can also write your own user defined function (UDF) (p. 262). For more information, see [How do I resolve the "function not registered" syntax error in 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. 138)` 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 on troubleshooting workgroup issues, see Troubleshooting Workgroups (p. 436).

Additional Resources

The following pages provide additional information for troubleshooting issues with Amazon Athena.

- Service Quotas (p. 558)
- Considerations and Limitations for SQL Queries in Amazon Athena (p. 528)
- Unsupported DDL (p. 503)
- Names for Tables, Databases, and Columns (p. 106)
- Data Types in Amazon Athena (p. 472)
- Supported SerDes and Data Formats (p. 134)
• Compression Formats (p. 160)
• Reserved Keywords (p. 107)
• Troubleshooting Workgroups (p. 436)

The following AWS resources can also be of help:

• Athena topics in the AWS Knowledge Center
• Athena discussion forum
• 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 visit the Amazon Athena Forum.
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

In general, Athena limits the runtime of each query to 30 minutes. Queries that run beyond this limit 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. 540)
- File Formats (p. 541)
- Joins, Grouping, and Unions (p. 541)
- Partitioning (p. 542)
- Window Functions (p. 542)
- Use More Efficient Functions (p. 542)

Data Size

Avoid single large files – Single files are loaded into a single node for processing. 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.bytecodeCompilationException 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 – Because query results are written to Amazon S3 by a single Athena node, a large amount of output data can slow performance. To work around this, try using CTAS (p. 514) to create a new table with the result of the query or INSERT INTO (p. 479) to append new results into an existing table.

Avoid CTAS queries with a large output – Because output data is written by a single node, 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 to a single node, 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. 186).

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. 248) 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. 506) to remove stale partitions.
- If your partition pattern is predictable, use partition projection (p. 119).

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. 506).

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. 119) to avoid the partition look up calls to AWS Glue.

Set reasonable partition projection properties – When using partition projection (p. 119), 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. 505). 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 on performance tuning in Athena, consider the following resources:

- Read the AWS Big Data blog post [Top 10 Performance Tuning Tips for Amazon Athena](#)
- Read other [Athena posts in the AWS Big Data Blog](#)
- Visit the [Amazon Athena Forum](#)
- Consult the [Athena topics in the AWS Knowledge Center](#)
- Contact AWS Support (in the AWS Management Console, click [Support, Support Center](#))
Code Samples, Service Quotas, and Previous JDBC Driver

Use code samples to create Athena applications based on AWS SDK for Java.

Use the links in this section to use earlier versions of the JDBC driver.

Learn about service quotas.

Topics
- Code Samples (p. 544)
- Using Earlier Version JDBC Drivers (p. 553)
- Service Quotas (p. 558)

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. 544)
  - Create a Client to Access Athena (p. 545)
  - Working with Query Executions
    - Start Query Execution (p. 545)
    - Stop Query Execution (p. 547)
    - List Query Executions (p. 549)
  - Working with Named Queries
    - Create a Named Query (p. 550)
    - Delete a Named Query (p. 551)
    - List Query Executions (p. 549)

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. 8) tutorial in Athena.

```java
package aws.example.athena;

public 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.

package aws.example.athena;
import software.amazon.awssdk.auth.credentials.InstanceProfileCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.athena.AthenaClient;
import software.amazon.awssdk.services.athena.AthenaClientBuilder;
public class AthenaClientFactory {
  private final AthenaClientBuilder builder = AthenaClient.builder()
    .region(Region.US_WEST_2)
    .credentialsProvider(InstanceProfileCredentialsProvider.create());
  public AthenaClient createClient() {
    return builder.build();
  }
}

Start Query Execution

The StartQueryExample shows how to submit a query to Athena, wait until the results become available, and then process the results.

package aws.example.athena;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.athena.AthenaClient;
import software.amazon.awssdk.services.athena.model.QueryExecutionContext;
import software.amazon.awssdk.services.athena.model.ResultConfiguration;
import software.amazon.awssdk.services.athena.model.StartQueryExecutionRequest;
import software.amazon.awssdk.services.athena.model.StartQueryExecutionResponse;
import software.amazon.awssdk.services.athena.model.AthenaException;
import software.amazon.awssdk.services.athena.model.GetQueryExecutionRequest;
import software.amazon.awssdk.services.athena.model.GetQueryExecutionResponse;
import software.amazon.awssdk.services.athena.model.QueryExecutionState;
import software.amazon.awssdk.services.athena.model.GetQueryResultsRequest;
import software.amazon.awssdk.services.athena.model.GetQueryResultsResponse;
import software.amazon.awssdk.services.athena.model.ColumnInfo;
import software.amazon.awssdk.services.athena.model.Row;
import software.amazon.awssdk.services.athena.model.Datum;
import software.amazon.awssdk.services.athena.paginators.GetQueryResultsIterable;
import java.util.List;
/**
*/
public class StartQueryExample {
    public static void main(String[] args) throws InterruptedException {
        AthenaClient athenaClient = AthenaClient.builder()
                .region(Region.US_WEST_2)
                .build();

        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(QueryExecutionStatus.COMPLETED.toString()) ||
                    queryState.equals(QueryExecutionStatus.FAILED.toString()) ||
                    queryState.equals(QueryExecutionStatus.CANCELLED.toString()) ||
                    queryState.equals(QueryExecutionStatus.TIMED_OUT.toString())) {
                isQueryStillRunning = false;
            } else {
                Thread.sleep(5000); // Wait for 5 seconds before checking again
            }
        }
    }

    // Process the result rows of the query
    public static void processResultRows(AthenaClient athenaClient, String queryExecutionId) {
        // Code to process result rows...
    }
}

* To run this Java V2 code example, ensure that you have setup your development environment, including your credentials.
* For information, see this documentation topic:
* https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
*/
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.
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.athena.AthenaClient;
import software.amazon.awssdk.services.athena.AthenaException;
import software.amazon.awssdk.services.athena.model.AthenaClientException;
import software.amazon.awssdk.services.athena.model.GetQueryExecutionRequest;
import software.amazon.awssdk.services.athena.model.GetQueryExecutionResponse;
import software.amazon.awssdk.services.athena.model.QueryExecutionState;
import software.amazon.awssdk.services.athena.model.StoppedQueryExecution;
import software.amazon.awssdk.services.athena.model.StartQueryExecutionRequest;
import software.amazon.awssdk.services.athena.model.StartQueryExecutionResponse;

/**
 * To run this Java V2 code example, ensure that you have setup your development environment, including your credentials.
 * For information, see this documentation topic:
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 */
public class StopQueryExecutionExample {
    public static void main(String[] args) throws Exception {
        AthenaClient athenaClient = AthenaClient.builder()
            .region(Region.US_WEST_2)
            .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().equals(QueryExecutionState.CANCELLED)) {
                System.out.println("The Amazon Athena query has been cancelled!");
            }
        } catch (AthenaException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}

548
// 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)
      .queryString(ExampleConstants.ATHENA_SAMPLE_QUERY)
      .resultConfiguration(resultConfiguration).build();

    StartQueryExecutionResponse startQueryExecutionResponse =
      athenaClient.startQueryExecution(startQueryExecutionRequest);
    return startQueryExecutionResponse.queryExecutionId();
  } catch (AthenaException e) {
    e.printStackTrace();
    System.exit(1);
  }
  return null;
}

List Query Executions

The ListQueryExecutionsExample shows how to obtain a list of query execution IDs.

package aws.example.athena;
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;

/**
 * To run this Java V2 code example, ensure that you have setup your development
 * environment, including your credentials.
 *
 * For information, see this 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) throws Exception {
    AthenaClient athenaClient = AthenaClient.builder()
      .region(Region.US_WEST_2)
      .build();

    listQueryIds(athenaClient);
    athenaClient.close();
  }
}
public static void listQueryIds(AthenaClient athenaClient) {
    try {
        ListQueryExecutionsRequest listQueryExecutionsRequest =
            ListQueryExecutionsRequest.builder().build();
        ListQueryExecutionsIterable listQueryExecutionResponses =
            athenaClient.listQueryExecutionsPaginator(listQueryExecutionsRequest);

        for (ListQueryExecutionsResponse listQueryExecutionResponse :
            listQueryExecutionResponses) {
            List<String> queryExecutionIds =
                listQueryExecutionResponse.queryExecutionIds();
            System.out.println("\n" + queryExecutionIds);
        }
    } 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.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;

/**
 * To run this Java V2 code example, ensure that you have setup your development
 * environment, including your credentials.
 *
 * For information, see this documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 */

public class CreateNamedQueryExample {
    public static void main(String[] args) throws Exception {
        final String USAGE = "\n" +
            "Usage:\n" +
            "  CreateNamedQueryExample <name>\n" +
            "  Where: name - the name of the Amazon Athena query.\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)
            .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);
    }
}

Delete a Named Query

The DeleteNamedQueryExample shows how to delete a named query by using the named query ID.

package aws.example.athena;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.athena.AthenaClient;
import software.amazon.awssdk.services.athena.model.DeleteNamedQueryRequest;
import software.amazon.awssdk.services.athena.model.AthenaException;
import software.amazon.awssdk.services.athena.model.CreateNamedQueryRequest;
import software.amazon.awssdk.services.athena.model.CreateNamedQueryResponse;

/**
 * To run this Java V2 code example, ensure that you have setup your development
 * environment, including your credentials.
 *
 * For information, see this documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 */
public class DeleteNamedQueryExample {
    public static void main(String[] args) {
        final String USAGE = "\n" +
            "Usage:\n" +
            "DeleteNamedQueryExample <name>\n" +
            "Where:\n" +
            "name - the name of the Amazon Athena query. \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)
            .build();
List Named Queries

The ListNamedQueryExample shows how to obtain a list of named query IDs.

```java
package aws.example.athena;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.athena.AthenaClient;
import software.amazon.awssdk.services.athena.AthenaException;
import software.amazon.awssdk.services.athena.AthenaClient;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;

/**
 * To run this Java V2 code example, ensure that you have setup your development
 * environment, including your credentials.
 *
 * For information, see this documentation topic:
 */
```
public class ListNamedQueryExample {
    public static void main(String[] args) throws Exception {
        AthenaClient athenaClient = AthenaClient.builder()
                .region(Region.US_WEST_2)
                .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);
        }
    }
}

Using Earlier Version JDBC Drivers

We recommend that you use the latest version of the JDBC driver. For information, see Using Athena with the JDBC Driver (p. 87). Links to earlier version 2.x drivers and support materials are below if required for your application.

Earlier Version JDBC Drivers

<table>
<thead>
<tr>
<th>JDBC Driver Version</th>
<th>Downloads</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC 4.2 and JDK 8.0 Compatible</td>
<td>AthenaJDBC42-2.0.8.jar Release Notes</td>
</tr>
<tr>
<td>JDBC 4.1 and JDK 7.0 Compatible</td>
<td>AthenaJDBC41-2.0.8.jar</td>
</tr>
</tbody>
</table>

License Agreement | Notices | Installation and Configuration Guide (PDF) | Migration Guide(PDF)
## Instructions for JDBC Driver version 1.1.0

This section includes a link to download version 1.1.0 of the JDBC driver. We highly recommend that you migrate to the current version of the driver. For information, see the JDBC Driver Migration Guide.

The JDBC driver version 1.0.1 and earlier versions are deprecated.

### Downloads

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0.7</td>
<td>JDBC 4.2 and JDK 8.0 Compatible – AthenaJDBC42-2.0.7.jar</td>
<td>License Agreement</td>
<td>Notices</td>
<td>Installation and Configuration Guide (PDF)</td>
<td>Migration Guide(PDF)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JDBC 4.1 and JDK 7.0 Compatible – AthenaJDBC41-2.0.7.jar</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.0.6</td>
<td>JDBC 4.2 and JDK 8.0 Compatible – AthenaJDBC42-2.0.6.jar</td>
<td>License Agreement</td>
<td>Notices</td>
<td>Installation and Configuration Guide (PDF)</td>
<td>Migration Guide(PDF)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JDBC 4.1 and JDK 7.0 Compatible – AthenaJDBC41-2.0.6.jar</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.0.5</td>
<td>JDBC 4.2 and JDK 8.0 Compatible – AthenaJDBC42-2.0.5.jar</td>
<td>License Agreement</td>
<td>Notices</td>
<td>Installation and Configuration Guide (PDF)</td>
<td>Migration Guide(PDF)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JDBC 4.1 and JDK 7.0 Compatible – AthenaJDBC41-2.0.5.jar</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.0.2</td>
<td>JDBC 4.2 and JDK 8.0 Compatible – AthenaJDBC42-2.0.2.jar</td>
<td>License Agreement</td>
<td>Notices</td>
<td>Installation and Configuration Guide (PDF)</td>
<td>Migration Guide(PDF)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JDBC 4.1 and JDK 7.0 Compatible – AthenaJDBC41-2.0.2.jar</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
JDBC driver version 1.1.0 is compatible with JDBC 4.1 and JDK 7.0. Use the following link to download the driver: AthenaJDBC41-1.1.0.jar. Also, download the driver license, and the third-party licenses for the driver. Use the AWS CLI with the following command: aws s3 cp s3://path_to_the_driver [local_directory], and then use the remaining instructions in this section.

**Note**
The following instructions are specific to JDBC version 1.1.0 and earlier.

### JDBC Driver Version 1.1.0: Specify the Connection String

To specify the JDBC driver connection URL in your custom application, use the string in this format:

```
jdbc:awsathena://athena.{REGION}.amazonaws.com:443
```

where `{REGION}` is a region identifier, such as `us-west-2`. For information on Athena regions see [Regions](#).

### JDBC Driver Version 1.1.0: Specify the JDBC Driver Class Name

To use the driver in custom applications, set up your Java class path to the location of the JAR file that you downloaded from Amazon S3 [https://s3.amazonaws.com/athena-downloads/drivers/JDBC/AthenaJDBC_1.1.0/AthenaJDBC41-1.1.0.jar](https://s3.amazonaws.com/athena-downloads/drivers/JDBC/AthenaJDBC_1.1.0/AthenaJDBC41-1.1.0.jar). This makes the classes within the JAR available for use. The main JDBC driver class is `com.amazonaws.athena.jdbc.AthenaDriver`.

### JDBC Driver Version 1.1.0: Provide the JDBC Driver Credentials

To gain access to AWS services and resources, such as Athena and the Amazon S3 buckets, provide JDBC driver credentials to your application.

To provide credentials in the Java code for your application:

1. Use a class which implements the `AWSCredentialsProvider`.
2. Set the JDBC property, `aws_credentials_provider_class`, equal to the class name, and include it in your classpath.
3. To include constructor parameters, set the JDBC property `aws_credentials_provider_arguments` as specified in the following section about configuration options.

Another method to supply credentials to BI tools, such as SQL Workbench, is to supply the credentials used for the JDBC as AWS access key and AWS secret key for the JDBC properties for user and password, respectively.

Users who connect through the JDBC driver and have custom access policies attached to their profiles need permissions for policy actions in addition to those in the [Amazon Athena API Reference](#).

### Policies for the JDBC Driver Version 1.1.0

You must allow JDBC users to perform a set of policy-specific actions. If the following actions are not allowed, users will be unable to see databases and tables:

- `athena:GetCatalogs`
- `athena:GetExecutionEngine`
- `athena:GetExecutionEngines`
- `athena:GetNamespace`
- `athena:GetNamespaces`
### JDBC Driver Version 1.1.0: Configure the JDBC Driver Options

You can configure the following options for the version of the JDBC driver version 1.1.0. With this version of the driver, you can also pass parameters using the standard JDBC URL syntax, for example: `jdbc:awsathena://athena.us-west-1.amazonaws.com:443?max_error_retries=20&connection_timeout=20000`.

#### Options for the JDBC Driver Version 1.0.1

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Default Value</th>
<th>Is Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>s3_staging_dir</td>
<td>The S3 location to which your query output is written, for example <code>s3://query-results-bucket/folder/</code></td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>query_results_encryption_option</td>
<td>The encryption method to use for the directory specified by <code>s3_staging_dir</code>. If not specified, the location is not encrypted. Valid values are SSE_S3, SSE_KMS, and CSE_KMS.</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>query_results_aws_kms_key</td>
<td>The Key ID of the AWS customer master key (CMK) to use if <code>query_results_encryption_option</code> specifies SSE-KMS or CSE-KMS. For example, 123abcde-4e56-56f7-g890-1234h5678i9j.</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>aws_credentials_provider_class</td>
<td>The credentials provider class name, which implements the AWSCredentialsProvider interface.</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>aws_credentials_provider_arguments</td>
<td>Arguments for the credentials provider constructor as comma-separated values.</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>max_error_retries</td>
<td>The maximum number of retries that the JDBC client attempts to make a request to Athena.</td>
<td>10</td>
<td>No</td>
</tr>
<tr>
<td>connection_timeout</td>
<td>The maximum amount of time, in milliseconds, to make a successful connection to Athena before an attempt is terminated.</td>
<td>10,000</td>
<td>No</td>
</tr>
<tr>
<td>socket_timeout</td>
<td>The maximum amount of time, in milliseconds, to wait for a socket in order to send data to Athena.</td>
<td>10,000</td>
<td>No</td>
</tr>
<tr>
<td>retry_base_delay</td>
<td>Minimum delay amount, in milliseconds, between retrying attempts to connect Athena.</td>
<td>100</td>
<td>No</td>
</tr>
<tr>
<td>retry_max_backoff_time</td>
<td>Maximum delay amount, in milliseconds, between retrying attempts to connect to Athena.</td>
<td>1000</td>
<td>No</td>
</tr>
<tr>
<td>log_path</td>
<td>Local path of the Athena JDBC driver logs. If no log path is provided, then no log files are created.</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>Property Name</td>
<td>Description</td>
<td>Default Value</td>
<td>Is Required</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>log_level</td>
<td>Log level of the Athena JDBC driver logs. Valid values: INFO, DEBUG, WARN, ERROR, ALL, OFF, FATAL, TRACE.</td>
<td>N/A</td>
<td>No</td>
</tr>
</tbody>
</table>

Examples: Using the 1.1.0 Version of the JDBC Driver with the JDK

The following code examples demonstrate how to use the JDBC driver version 1.1.0 in a Java application. These examples assume that the AWS JAVA SDK is included in your classpath, specifically the aws-java-sdk-core module, which includes the authorization packages (com.amazonaws.auth.*) referenced in the examples.

**Example Example: Creating a Driver Version 1.0.1**

```java
    Properties info = new Properties();
    info.put("user", "AWSAccessKey");
    info.put("password", "AWSSecretAccessKey");
    info.put("s3_staging_dir", "s3://S3 Bucket Location/");
    info.put("aws_credentials_provider_class", "com.amazonaws.auth.DefaultAWSCredentialsProviderChain");
    Class.forName("com.amazonaws.athena.jdbc.AthenaDriver");
    Connection connection = DriverManager.getConnection("jdbc:awsathena://athena.us-east-1.amazonaws.com:443/", info);
```

The following examples demonstrate different ways to use a credentials provider that implements the AWSCredentialsProvider interface with the previous version of the JDBC driver.

**Example Example: Using a Credentials Provider for JDBC Driver 1.0.1**

```java
    Properties myProps = new Properties();
    myProps.put("aws_credentials_provider_class", "com.amazonaws.auth.PropertiesFileCredentialsProvider");
    myProps.put("aws_credentials_provider_arguments", "/Users/myUser/.athenaCredentials");
```

In this case, the file `/Users/myUser/.athenaCredentials` should contain the following:

```plaintext
accessKey = ACCESSKEY
secretKey = SECRETKEY
```

Replace the right part of the assignments with your account's AWS access and secret keys.

**Example Example: Using a Credentials Provider with Multiple Arguments**

This example shows an example credentials provider, CustomSessionsCredentialsProvider, that uses an access and secret key in addition to a session token. CustomSessionsCredentialsProvider is shown for example only and is not included in the driver. The signature of the class looks like the following:
public CustomSessionsCredentialsProvider(String accessId, String secretKey, String token) {
    //...
}

You would then set the properties as follows:

Properties myProps = new Properties();
myProps.put("aws_credentials_provider_class","com.amazonaws.athena.jdbc.CustomSessionsCredentialsProvider");
String providerArgs = "My_Access_Key," + "My_Secret_Key," + "My_Token";
myProps.put("aws_credentials_provider_arguments",providerArgs);

Note
If you use the InstanceProfileCredentialsProvider, you don't need to supply any credential provider arguments because they are provided using the Amazon EC2 instance profile for the instance on which you are running your application. You would still set the aws_credentials_provider_class property to this class name, however.

Policies for the JDBC Driver Earlier than Version 1.1.0

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>

Service Quotas

Note
The Service Quotas console provides information about Amazon Athena quotas. Along with viewing the default quotas, you can use the Service Quotas console to request quota increases for the quotas that are adjustable.

Queries

Your account has the following default query-related quotas per AWS Region for Amazon Athena:

- **DDL query quota** – 20 DDL active queries. DDL queries include CREATE TABLE and ALTER TABLE ADD PARTITION queries.
- **DDL query timeout** – The DDL query timeout is 600 minutes.
- **DML query quota** – 25 DML active queries in the US East (N. Virginia) Region; 20 DML active queries in all other Regions. DML queries include SELECT and CREATE TABLE AS (CTAS) queries.
• **DML query timeout** – The DML query timeout is 30 minutes.

These are soft quotas; you can use the [Athena Service Quotas](https://aws.amazon.com/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 you are using the default DML quota and your total of running and queued queries exceeds 25, query 26 will result in a "too many queries" 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?](https://aws.amazon.com/athena/support/) 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 tags per workgroup is 50. For more information, see [Tag Restrictions](p. 448).

### Databases, Tables, and Partitions

• If you are using the AWS Glue Data Catalog with Athena, see [AWS Glue Endpoints and Quotas](https://aws.amazon.com/athena/endpoints/) for service quotas on tables, databases, and partitions – for example, the maximum number of databases or tables per account.
• 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, ListNamedQueries, ListQueryExecutions</td>
<td>5</td>
<td>up to 10</td>
</tr>
<tr>
<td>CreateNamedQuery, DeleteNamedQuery, GetNamedQuery</td>
<td>5</td>
<td>up to 20</td>
</tr>
<tr>
<td>BatchGetQueryExecution</td>
<td>20</td>
<td>up to 40</td>
</tr>
<tr>
<td>StartQueryExecution, StopQueryExecution</td>
<td>20</td>
<td>up to 80</td>
</tr>
<tr>
<td>GetQueryExecution, GetQueryResults</td>
<td>100</td>
<td>up to 200</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.
Release Notes

Describes Amazon Athena features, improvements, and bug fixes by release date.

Release Dates

- August 12, 2021 (p. 562)
- August 6, 2021 (p. 563)
- August 5, 2021 (p. 563)
- July 30, 2021 (p. 563)
- July 21, 2021 (p. 563)
- July 16, 2021 (p. 564)
- July 8, 2021 (p. 564)
- July 1, 2021 (p. 564)
- June 23, 2021 (p. 565)
- May 12, 2021 (p. 565)
- May 10, 2021 (p. 565)
- May 5, 2021 (p. 565)
- April 30, 2021 (p. 566)
- April 29, 2021 (p. 566)
- April 26, 2021 (p. 566)
- April 21, 2021 (p. 566)
- April 5, 2021 (p. 566)
- March 30, 2021 (p. 567)
- March 25, 2021 (p. 567)
- March 5, 2021 (p. 567)
- February 25, 2021 (p. 567)
- December 16, 2020 (p. 568)
- November 24, 2020 (p. 568)
- November 11, 2020 (p. 568)
- October 22, 2020 (p. 570)
- July 29, 2020 (p. 570)
- July 9, 2020 (p. 570)
- June 1, 2020 (p. 571)
- May 21, 2020 (p. 571)
- April 1, 2020 (p. 571)
- March 11, 2020 (p. 571)
- March 6, 2020 (p. 572)
- November 26, 2019 (p. 572)
- November 12, 2019 (p. 575)
- November 8, 2019 (p. 575)
- October 8, 2019 (p. 575)
- September 19, 2019 (p. 575)
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. 88).
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. 482).

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.

  ```
  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. 515) 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. 88).

For more information, see Using the Amazon Athena Power BI Connector (p. 90).

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. 248).

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. 88).

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. 484) statements and parameterized queries (p. 195).

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. 87) and Connecting to Amazon Athena with ODBC (p. 88).

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. 22) and Cross-Account Access to Data Catalogs (p. 341).

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. 88).

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 on GitHub.
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 Querying with Prepared Statements (p. 195).

To download the new drivers, release notes, and documentation, see Using Athena with the JDBC Driver (p. 87) and Connecting to Amazon Athena with ODBC (p. 88).

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. 463).

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. 463).

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 the EXPLAIN Statement in Athena (p. 484) and Understanding Athena EXPLAIN Statement Results (p. 490).

**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. 260).

**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. 262).

**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. 463).

**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. 463).

**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. 463).

**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. 463).

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. 463) and Using Amazon Athena Federated Query (p. 68).

**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. 371).

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. 87) and Connecting to Amazon Athena with ODBC (p. 88).

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. 463).
- For information about how to upgrade, see Changing Athena Engine Versions (p. 459).
- For information about testing queries, see Testing Queries in Advance of an Engine Version Upgrade (p. 463).

### Federated SQL Queries

You can now use Athena’s federated query in the 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. 73)
- CloudWatch Metrics (p. 73)
- DocumentDB (p. 74)
- DynamoDB (p. 74)
- Elasticsearch (p. 74)
- HBase (p. 74)
- JDBC-compliant data sources (p. 74)
- Neptune (p. 75)
- Redis (p. 75)
- Timestream (p. 75)
- TPC Benchmark DS (TPC-DS) (p. 75)
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. 68).
- 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. 70), Okta service for authentication, and connections to VPC endpoints.

To download and use the new version of the driver, see Using Athena with the JDBC Driver (p. 87).

July 9, 2020

Published on 2020-07-09

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. 248).

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.

June 1, 2020

Published on 2020-06-01

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. 67) 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. 36).

May 21, 2020

Published on 2020-05-21

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. 119).

April 1, 2020

Published on 2020-04-01

In addition to the US East (N. Virginia) Region, the Amazon Athena federated query (p. 68), user defined functions (UDFs) (p. 262), machine learning inference (p. 260), and external Hive metastore (p. 36) 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. 442).

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.

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
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 as 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. 372). 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. 88). 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
Driver;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. 435) to create and manage workgroups, and Athena API tag actions (p. 450) 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. 420) and Workgroup Example Policies (p. 424).
- Tagging Resources (p. 448) and Tag-Based IAM Access Control Policies (p. 453).

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. 87).

February 22, 2019

Published on 2019-02-22

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 Workgroups (p. 448).

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 Working with Tags Using the API Actions (p. 448).

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. 455).

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. 87).

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. 420). 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. 420) and Controlling Costs and Monitoring Queries with CloudWatch Metrics and Events (p. 437).

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. 423), and the AmazonAthenaFullAccess Managed Policy (p. 326).

You can start using workgroups in the console, with the workgroup API operations (p. 435), or with the JDBC driver. For a high-level procedure, see Setting up Workgroups (p. 422). To download the JDBC driver with workgroup support, see Using Athena with the JDBC Driver (p. 87).

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=<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. 435) and Troubleshooting Workgroups (p. 436).

- 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. 147).

• 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. 100) and Transitioning to the GLACIER Storage Class (Object Archival) in the Amazon Simple Storage Service Developer 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. 290).

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. 362).

To download the JDBC driver version 2.0.6 and its documentation, see Using Athena with the JDBC Driver (p. 87). 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. 88). 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. 344).
• 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. 335).

Note
Data resides in the Amazon S3 buckets, and access to it is governed by the Amazon S3 Permissions (p. 331). 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. 177).

Use CTAS queries to:

• Create a table from query results (p. 177) in one step.
• Create CTAS queries in the Athena console (p. 179), using Examples (p. 183). For information about syntax, see CREATE TABLE AS (p. 514).
• 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. 177) and Columnar Storage Formats (p. 110).

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. 88).

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 AWSQuicksightAthenaAccess Managed Policy (p. 328).

**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 (p. 472).

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. 87).

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 AWSQuicksightAthenaAccess Managed Policy (p. 328). 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 Using the Previous Version of the JDBC Driver (p. 553), and the JDBC Driver Migration Guide.

August 7, 2018

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. 294) and Amazon VPC Flow Logs can now be delivered to S3.

June 5, 2018

Topics

• Support for Views (p. 582)
• Improvements and Updates to Error Messages (p. 582)
• Bug Fixes (p. 583)

Support for Views

Added support for views. You can now use CREATE VIEW (p. 516), DESCRIBE VIEW (p. 518), DROP VIEW (p. 519), SHOW CREATE VIEW (p. 523), and SHOW VIEWS (p. 527) 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. 172).

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. 197).
• 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

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. 558). 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. 87).

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.

For information about the previous version of the JDBC driver, see Using Athena with the Previous Version of the JDBC Driver (p. 553).

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. 277).

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. 150) and OpenCSV SerDe (p. 140), 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 SQL Queries, Functions, and Operators (p. 473) in this guide, and Presto 0.172 Functions.

Athena does not support all of Presto's features. For more information, see Limitations (p. 528).

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. 88).

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. 223) 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 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. 18).

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. 143). 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. 87).
• 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. 318).

**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. For more information, see Using Athena with the Previous Version of the JDBC Driver (p. 553) for the previous version of the driver, and Connect with the JDBC (p. 87), for the most current version.

• 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. For more information, see CloudTrail SerDe (p. 138). For detailed usage examples, see the AWS Big Data Blog post, Analyze Security, Compliance, and Operational Activity Using AWS CloudTrail and Amazon Athena.

**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. 136)
  - OpenCSVserDe for Processing CSV (p. 140)
- **US East (Ohio) Region (us-east-2) launch.** You can now run queries in this region.
- You can now use the Add Table wizard to define table schema in bulk. Choose Catalog Manager, Add table, and then choose Bulk add columns as you walk through the steps to define the table.
Type name value pairs in the text box and choose **Add**.

**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: August 19, 2021.**

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 UNLOAD statement.</td>
<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. 482).</td>
<td>August 5, 2021</td>
</tr>
<tr>
<td>Added documentation for the Athena EXPLAIN statement feature.</td>
<td>For more information, see Using the EXPLAIN Statement in Athena (p. 484) and Understanding Athena EXPLAIN Statement Results (p. 490).</td>
<td>April 5, 2021</td>
</tr>
<tr>
<td>Added pages on troubleshooting and performance tuning in Athena.</td>
<td>For more information, see Troubleshooting in Athena (p. 530) and Performance Tuning in Athena (p. 540).</td>
<td>December 30, 2020</td>
</tr>
<tr>
<td>Added documentation for Athena engine versioning and Athena engine version 2.</td>
<td>For more information, see Athena Engine Versioning (p. 459).</td>
<td>November 11, 2020</td>
</tr>
<tr>
<td>Updated federated query documentation for general availability release.</td>
<td>For more information, see Using Amazon Athena Federated Query (p. 68) and Using Athena with CalledVia Context Keys (p. 346).</td>
<td>November 11, 2020</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. 380) and Tutorial: Configuring Federated Access for Okta Users to Athena Using Lake Formation and JDBC (p. 381).</td>
<td>September 25, 2020</td>
</tr>
<tr>
<td>Added documentation for the Amazon Athena Elasticsearch data connector.</td>
<td>For more information, see Amazon Athena Elasticsearch Connector (p. 74).</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. 248).</td>
<td>July 9, 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 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. 307) and [Querying Internet Information Server (IIS) Logs Stored in Amazon S3](p. 309).</td>
<td>July 8, 2020</td>
</tr>
<tr>
<td>The Amazon Athena User Guide is now available in Kindle format.</td>
<td>The Kindle ebook is free of charge. For more information, see [Amazon Athena: User Guide Kindle Edition](p. 592), or choose the Kindle link at the top of any page in the online version of the Amazon Athena User Guide.</td>
<td>June 18, 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. 36).</td>
<td>June 1, 2020</td>
</tr>
<tr>
<td>Added documentation for tagging data catalog resources.</td>
<td>For more information, see [Tagging Resources](p. 448).</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. 119).</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. 544).</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. 287).</td>
<td>March 19, 2020</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. 442).</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. 285).</td>
<td>February 6, 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 documentation on using CTAS with INSERT INTO to add data from an unpartitioned source to a partitioned destination. | Documentation updates include, but are not limited to, the following topics:  
  - Using CTAS and INSERT INTO for ETL and Data Analysis (p. 186)  
  - Connecting to Amazon Athena with ODBC (p. 88) (The 1.1.0 preview features are now included in the 1.1.2 ODBC driver.)  
  - SHOW DATABASES (p. 523)  
  - CREATE TABLE AS (p. 514)  
For more information, see Using CTAS and INSERT INTO to Create a Table with More Than 100 Partitions (p. 192). | February 4, 2020 |
| Added download links for the 1.1.0 preview version of the ODBC driver for Athena. |                                                                                                                                             | January 22, 2020 |
| Description for SHOW DATABASES LIKE regex corrected.                    | For more information, see Specifying a Query Result Location (p. 167).                                                                                                                                   | January 20, 2020 |
| Corrected partitioned_by syntax in CTA topic.                          | For more information, see the following topics:  
  - Querying AWS Glue Data Catalog (p. 303)  
  - Service Quotas (p. 558)  
For more information, see OpenCSVSerDe for Processing CSV (p. 140). | January 17, 2020 |
<p>| Other minor fixes.                                                      |                                                                                                                                             | January 15, 2020 |
| Query results location information updated.                            |                                                                                                                                             | January 20, 2020 |
| Added topic on querying the AWS Glue Data Catalog.                     | Athena no longer creates a 'default' query results location. For more information, see Specifying a Query Result Location (p. 167).                                                                    | January 20, 2020 |
| Corrected topic on OpenCSVSerDe to note that the TIMESTAMP type should be specified in the UNIX numeric format. |                                                                                                                                             | January 15, 2020 |</p>
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Release Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Updated security topic on encryption to note that Athena does not support asymmetric keys.</td>
<td>Athena supports only symmetric keys for reading and writing data. For more information, see Supported Amazon S3 Encryption Options (p. 318).</td>
<td>January 8, 2020</td>
</tr>
<tr>
<td>Added information on cross-account access to an Amazon S3 buckets that are encrypted with a custom AWS KMS key.</td>
<td>For more information, see Cross-account Access to a Bucket Encrypted with a Custom AWS KMS Key (p. 332).</td>
<td>December 13, 2019</td>
</tr>
</tbody>
</table>
| 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. 68)  
- Using Athena Data Source Connectors (p. 72)  
- Using Athena Data Connector for External Hive Metastore (p. 36)  
- Using Machine Learning (ML) with Amazon Athena (p. 260)  
- Querying with User Defined Functions (p. 262)  
- List of CloudWatch Metrics and Dimensions for Athena (p. 441) | 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. 479) and Working with Query Results, Output Files, and Query History (p. 162). | September 18, 2019 |
| Added section for interface VPC endpoints (PrivateLink) support. Updated JDBC drivers. Updated information on enriched VPC flow logs. | For more information, see Connect to Amazon Athena Using an Interface VPC Endpoint (p. 371), Querying Amazon VPC Flow Logs (p. 294), and Using Athena with the JDBC Driver (p. 87). | September 11, 2019 |
| Added section on integrating with AWS Lake Formation. | For more information, see Using Athena to Query Data Registered With AWS Lake Formation (p. 372). | June 26, 2019 |
| Updated Security section for consistency with other AWS services. | For more information, see Amazon Athena Security (p. 316). | June 26, 2019 |</p>
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Release Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Added section on querying AWS WAF logs.</td>
<td>For more information, see Querying AWS WAF Logs (p. 296).</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. 88). 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. 435) to create and manage workgroups, and Athena API tag actions (p. 450) 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>
</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. 147). | February 18, 2019 |
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Release Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Added support for workgroups.</td>
<td>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. 420) and Controlling Costs and Monitoring Queries with CloudWatch Metrics and Events (p. 437).</td>
<td>February 18, 2019</td>
</tr>
<tr>
<td>Added support for analyzing logs from Network Load Balancer.</td>
<td>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. 290).</td>
<td>January 24, 2019</td>
</tr>
<tr>
<td>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).</td>
<td>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. 362).</td>
<td>November 10, 2018</td>
</tr>
<tr>
<td>Added support for fine-grained access control to databases and tables in Athena. Additionally, added policies in Athena that allow you to encrypt database and table metadata in the Data Catalog.</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. Additionally, you can encrypt database and table metadata in the Data Catalog, by adding specific policies to Athena. For details, see Fine-Grained Access to Databases and Tables in the AWS Glue Data Catalog (p. 335).</td>
<td>October 15, 2018</td>
</tr>
<tr>
<td>Added support for CREATE TABLE AS SELECT statements. Made other improvements in the documentation.</td>
<td>Added support for CREATE TABLE AS SELECT statements. See Creating a Table from Query Results (p. 177), Considerations and Limitations (p. 177), and Examples (p. 183).</td>
<td>October 10, 2018</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Release Date</td>
</tr>
<tr>
<td>-----------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Released the ODBC driver version 1.0.3 with support for streaming results instead of fetching them in pages. Made other improvements in the documentation.</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;. For downloading the ODBC driver version 1.0.3 and its documentation, see Connecting to Amazon Athena with ODBC (p. 88).</td>
<td>September 6, 2018</td>
</tr>
<tr>
<td>Released the JDBC driver version 2.0.5 with default support for streaming results instead of fetching them in pages. Made other improvements in the documentation.</td>
<td>Released the JDBC driver 2.0.5 with default support for streaming results instead of fetching them in pages. For information, see Using Athena with the JDBC Driver (p. 87).</td>
<td>August 16, 2018</td>
</tr>
<tr>
<td>Updated the documentation for querying Amazon Virtual Private Cloud flow logs, which can be stored directly in Amazon S3 in a GZIP format. Updated examples for querying ALB logs.</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 Querying Amazon VPC Flow Logs (p. 294). Updated examples for querying ALB logs. For information, see Querying Application Load Balancer Logs (p. 271).</td>
<td>August 7, 2018</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. 172). Updated this guide with guidance on handling schema updates for various data storage formats. For information, see Handling Schema Updates (p. 197).</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. 558).</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. 17).</td>
<td>May 8, 2018</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Release Date</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>--------------</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. 87).</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. 277).</td>
<td>March 15, 2018</td>
</tr>
<tr>
<td>Added support for securely offloading intermediate data to disk for queries with <code>GROUP BY</code>.</td>
<td>Added an ability to securely offload intermediate data to disk for memory-intensive queries that use the <code>GROUP BY</code> 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. 584).</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. 584).</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>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 <code>AWS::Athena::NamedQuery</code> 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>Change</td>
<td>Description</td>
<td>Release Date</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>--------------</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. 270) and for querying different types of data in Athena. For information, see Running SQL Queries Using Amazon Athena (p. 162).</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. 318).</td>
<td>April 4, 2017</td>
</tr>
<tr>
<td>Added the AWS CloudTrail SerDe.</td>
<td>Added the AWS CloudTrail SerDe, improved performance, fixed partition issues. For more information, see CloudTrail SerDe (p. 138).</td>
<td>March 24, 2017</td>
</tr>
<tr>
<td>• Improved performance when scanning a large number of partitions. • Improved performance on <code>MSCK Repair Table</code> 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.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Added support for US East (Ohio).</td>
<td>Added support for Avro SerDe (p. 136) and OpenCSVSerDe for Processing CSV (p. 140), US East (Ohio), and bulk editing columns in the console wizard. Improved performance on large Parquet tables.</td>
<td>February 20, 2017</td>
</tr>
<tr>
<td>The initial release of the Amazon Athena User Guide.</td>
<td></td>
<td>November, 2016</td>
</tr>
</tbody>
</table>
AWS glossary

For the latest AWS terminology, see the AWS glossary in the AWS General Reference.