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

## Release Notes

<table>
<thead>
<tr>
<th>Date</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>March 05, 2019</td>
<td>7</td>
</tr>
<tr>
<td>February 22, 2019</td>
<td>7</td>
</tr>
<tr>
<td>February 18, 2019</td>
<td>8</td>
</tr>
<tr>
<td>November 20, 2018</td>
<td>9</td>
</tr>
<tr>
<td>October 15, 2018</td>
<td>9</td>
</tr>
<tr>
<td>October 10, 2018</td>
<td>10</td>
</tr>
<tr>
<td>September 6, 2018</td>
<td>10</td>
</tr>
<tr>
<td>August 23, 2018</td>
<td>11</td>
</tr>
<tr>
<td>August 16, 2018</td>
<td>11</td>
</tr>
<tr>
<td>August 7, 2018</td>
<td>12</td>
</tr>
<tr>
<td>June 5, 2018</td>
<td>12</td>
</tr>
<tr>
<td>May 17, 2018</td>
<td>13</td>
</tr>
<tr>
<td>April 19, 2018</td>
<td>13</td>
</tr>
<tr>
<td>April 6, 2018</td>
<td>14</td>
</tr>
<tr>
<td>March 15, 2018</td>
<td>14</td>
</tr>
<tr>
<td>February 2, 2018</td>
<td>14</td>
</tr>
<tr>
<td>January 19, 2018</td>
<td>14</td>
</tr>
<tr>
<td>November 13, 2017</td>
<td>15</td>
</tr>
<tr>
<td>November 1, 2017</td>
<td>15</td>
</tr>
<tr>
<td>October 19, 2017</td>
<td>15</td>
</tr>
<tr>
<td>October 3, 2017</td>
<td>15</td>
</tr>
<tr>
<td>September 25, 2017</td>
<td>15</td>
</tr>
<tr>
<td>August 14, 2017</td>
<td>16</td>
</tr>
<tr>
<td>August 4, 2017</td>
<td>16</td>
</tr>
<tr>
<td>June 22, 2017</td>
<td>16</td>
</tr>
<tr>
<td>June 8, 2017</td>
<td>16</td>
</tr>
<tr>
<td>May 19, 2017</td>
<td>16</td>
</tr>
<tr>
<td>Improvements</td>
<td>17</td>
</tr>
<tr>
<td>Bug Fixes</td>
<td>17</td>
</tr>
<tr>
<td>April 4, 2017</td>
<td>17</td>
</tr>
<tr>
<td>Features</td>
<td>17</td>
</tr>
<tr>
<td>Improvements</td>
<td>18</td>
</tr>
<tr>
<td>Bug Fixes</td>
<td>18</td>
</tr>
<tr>
<td>March 24, 2017</td>
<td>18</td>
</tr>
<tr>
<td>Features</td>
<td>18</td>
</tr>
<tr>
<td>Improvements</td>
<td>18</td>
</tr>
<tr>
<td>Bug Fixes</td>
<td>19</td>
</tr>
<tr>
<td>February 20, 2017</td>
<td>19</td>
</tr>
<tr>
<td>Features</td>
<td>19</td>
</tr>
<tr>
<td>Improvements</td>
<td>20</td>
</tr>
</tbody>
</table>

## Setting Up

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sign Up for AWS</td>
<td>21</td>
</tr>
<tr>
<td>To create an AWS account</td>
<td>21</td>
</tr>
<tr>
<td>Create an IAM User</td>
<td>21</td>
</tr>
<tr>
<td>To create a group for administrators</td>
<td>21</td>
</tr>
</tbody>
</table>
# Amazon Athena User Guide

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions Supported</td>
<td>68</td>
</tr>
<tr>
<td>Transactional Data Transformations Are Not Supported</td>
<td>68</td>
</tr>
<tr>
<td>Operations That Change Table States Are ACID</td>
<td>68</td>
</tr>
<tr>
<td>All Tables Are EXTERNAL</td>
<td>69</td>
</tr>
<tr>
<td>UDF and UDAF Are Not Supported</td>
<td>69</td>
</tr>
<tr>
<td>To create a table using the AWS Glue Data Catalog</td>
<td>69</td>
</tr>
<tr>
<td>To create a table using the wizard</td>
<td>69</td>
</tr>
<tr>
<td>To create a database using Hive DDL</td>
<td>69</td>
</tr>
<tr>
<td>To create a table using Hive DDL</td>
<td>70</td>
</tr>
<tr>
<td>Names for Tables, Databases, and Columns</td>
<td>71</td>
</tr>
<tr>
<td>Table names and table column names in Athena must be lowercase</td>
<td>71</td>
</tr>
<tr>
<td>Athena table, view, database, and column names allow only underscore special characters</td>
<td>71</td>
</tr>
<tr>
<td>Names that begin with an underscore</td>
<td>71</td>
</tr>
<tr>
<td>Table or view names that include numbers</td>
<td>71</td>
</tr>
<tr>
<td>Reserved Keywords</td>
<td>72</td>
</tr>
<tr>
<td>List of Reserved Keywords in DDL Statements</td>
<td>72</td>
</tr>
<tr>
<td>List of Reserved Keywords in SQL SELECT Statements</td>
<td>72</td>
</tr>
<tr>
<td>Examples of Queries with Reserved Words</td>
<td>73</td>
</tr>
<tr>
<td>Table Location in Amazon S3</td>
<td>73</td>
</tr>
<tr>
<td>Partitioning Data</td>
<td>74</td>
</tr>
<tr>
<td>Scenario 1: Data already partitioned and stored on S3 in hive format</td>
<td>74</td>
</tr>
<tr>
<td>Scenario 2: Data is not partitioned</td>
<td>76</td>
</tr>
<tr>
<td>Columnar Storage Formats</td>
<td>78</td>
</tr>
<tr>
<td>Converting to Columnar Formats</td>
<td>78</td>
</tr>
<tr>
<td>Overview</td>
<td>78</td>
</tr>
<tr>
<td>Before you begin</td>
<td>23</td>
</tr>
<tr>
<td>Example: Converting data to Parquet using an EMR cluster</td>
<td>80</td>
</tr>
<tr>
<td>Querying Data in Amazon Athena Tables</td>
<td>83</td>
</tr>
<tr>
<td>Query Results</td>
<td>83</td>
</tr>
<tr>
<td>Saving Query Results</td>
<td>84</td>
</tr>
<tr>
<td>The Default Location for Query Results</td>
<td>84</td>
</tr>
<tr>
<td>Viewing Query History</td>
<td>85</td>
</tr>
<tr>
<td>Views</td>
<td>86</td>
</tr>
<tr>
<td>When to Use Views?</td>
<td>86</td>
</tr>
<tr>
<td>Supported Actions for Views in Athena</td>
<td>87</td>
</tr>
<tr>
<td>Considerations for Views</td>
<td>87</td>
</tr>
<tr>
<td>Working with Views in the Console</td>
<td>88</td>
</tr>
<tr>
<td>Creating Views</td>
<td>89</td>
</tr>
<tr>
<td>Examples of Views</td>
<td>90</td>
</tr>
<tr>
<td>Updating Views</td>
<td>91</td>
</tr>
<tr>
<td>Deleting Views</td>
<td>91</td>
</tr>
<tr>
<td>Creating a Table from Query Results (CTAS)</td>
<td>91</td>
</tr>
<tr>
<td>Considerations and Limitations for CTAS Queries</td>
<td>91</td>
</tr>
<tr>
<td>Running CTAS Queries in the Console</td>
<td>93</td>
</tr>
<tr>
<td>Bucketing vs Partitioning</td>
<td>96</td>
</tr>
<tr>
<td>Examples of CTAS Queries</td>
<td>97</td>
</tr>
<tr>
<td>Querying Arrays</td>
<td>100</td>
</tr>
<tr>
<td>Creating Arrays</td>
<td>100</td>
</tr>
<tr>
<td>Concatenating Arrays</td>
<td>102</td>
</tr>
<tr>
<td>Converting Array Data Types</td>
<td>103</td>
</tr>
<tr>
<td>Finding Lengths</td>
<td>103</td>
</tr>
<tr>
<td>Accessing Array Elements</td>
<td>103</td>
</tr>
<tr>
<td>Flattening Nested Arrays</td>
<td>104</td>
</tr>
<tr>
<td>Creating Arrays from Subqueries</td>
<td>107</td>
</tr>
<tr>
<td>Filtering Arrays</td>
<td>107</td>
</tr>
<tr>
<td>Sorting Arrays</td>
<td>108</td>
</tr>
</tbody>
</table>
Handling Schema Updates ............................................................................................................... 148
Querying AWS Service Logs ............................................................................................................. 135
Querying Geospatial Data ................................................................................................................ 123
Summary: Updates and Data Formats in Athena ......................................................................... 148
Querying Amazon VPC Flow Logs ............................................................................................. 145
Querying Application Load Balancer Logs .................................................................................. 143
Querying Network Load Balancer Logs ...................................................................................... 142
Querying Amazon CloudFront Logs ........................................................................................... 139
Querying AWS CloudTrail Logs ................................................................................................. 135
Input Data Formats and Geometry Data Types ........................................................................... 123
Input Data Formats ...................................................................................................................... 123
Geometry Data Types .................................................................................................................... 124
List of Supported Geospatial Functions ..................................................................................... 124
Before You Begin ......................................................................................................................... 124
Constructor Functions .................................................................................................................. 125
Geospatial Relationship Functions ............................................................................................. 127
Operation Functions .................................................................................................................... 128
Accessor Functions ....................................................................................................................... 130
Examples: Geospatial Queries ..................................................................................................... 132
Querying AWS Service Logs ............................................................................................................. 135
Querying AWS CloudTrail Logs ................................................................................................. 135
Understanding CloudTrail Logs and Athena Tables ........................................................................ 136
Creating a Table for CloudTrail Logs in the CloudTrail Console .................................................. 136
Manually Creating the Table for CloudTrail Logs in Athena ............................................................ 137
Example Query for CloudTrail Logs ............................................................................................. 138
Tips for Querying CloudTrail Logs ............................................................................................... 138
Querying Amazon CloudFront Logs ........................................................................................... 139
Creating the Table for CloudFront Logs ..................................................................................... 139
Example Query for CloudFront Logs ............................................................................................ 140
querying Classic Load Balancer Logs ............................................................................................. 140
To create the table for Elastic Load Balancing logs ........................................................................ 141
Elastic Load Balancing Example Queries ...................................................................................... 141
Querying Network Load Balancer Logs ........................................................................................ 142
To create the table for Network Load Balancer logs ...................................................................... 142
Network Load Balancer Example Queries .................................................................................... 143
Querying Application Load Balancer Logs ................................................................................ 143
Creating the Table for ALB Logs .................................................................................................. 143
Example Queries for ALB Logs ...................................................................................................... 144
Querying Amazon VPC Flow Logs .............................................................................................. 145
Creating the Table for VPC Flow Logs ........................................................................................ 145
Example Queries for Amazon VPC Flow Logs ............................................................................. 146
Handling Schema Updates ............................................................................................................... 148
Summary: Updates and Data Formats in Athena ......................................................................... 148
Index Access in ORC and Parquet ................................................................................................... 149
ORC: Read by Index ....................................................................................................................... 150
Parquet: Read by Name .................................................................................................................. 150
Types of Updates ............................................................................................................................ 151
Adding Columns at the Beginning or in the Middle of the Table .................................................. 152
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. 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—executing queries in parallel—so results are fast, even with large datasets and complex queries.

Topics
- When should I use Athena? (p. 1)
- Accessing Athena (p. 1)
- Understanding Tables, Databases, and the Data Catalog (p. 2)
- AWS Service Integrations with Athena (p. 3)

When should I use Athena?

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

Athena integrates with Amazon QuickSight for easy data visualization. You can use Athena to generate reports or to explore data with business intelligence tools or SQL clients connected with a JDBC or an ODBC driver. For more information, see What is Amazon QuickSight in the Amazon QuickSight User Guide and Connecting to Amazon Athena with ODBC and JDBC Drivers (p. 43).

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 AWS account and integrated with the ETL and data discovery features of AWS Glue. For more information, see Integration with AWS Glue (p. 28) and What is AWS Glue in the AWS Glue Developer Guide.

For a list of AWS services that Athena leverages or integrates with, see the section called “AWS Service Integrations with Athena” (p. 3).

Accessing Athena

You can access Athena using the AWS Management Console, through a JDBC or ODBC connection, using the Athena API, or using the Athena CLI.

- To get started with the console, see Getting Started (p. 23).
- To learn how to use JDBC or ODBC drivers, see Connecting to Amazon Athena with JDBC (p. 43) and Connecting to Amazon Athena with ODBC (p. 44).
- To use the Athena API, see the Amazon Athena API 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 either in the AWS Glue Data Catalog, or in the internal Athena 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. 28). 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.

The AWS Glue Data Catalog is accessible throughout your AWS 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.

Note
You use the internal Athena data catalog in regions where AWS Glue is not available and where the AWS Glue Data Catalog cannot be used.

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 execute 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, or in its internal data catalog in those regions where AWS Glue is not available.

Note
If you have tables in Athena created before August 14, 2017, they were created in an Athena-managed 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. 29).

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. 23).
- 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 query data from other AWS services in Athena. Athena leverages several AWS services. For more information, see the following table.

**Note**
To see the list of supported regions for each service, see Regions and Endpoints in the Amazon Web Services General Reference.

<table>
<thead>
<tr>
<th>AWS Service</th>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS CloudTrail</td>
<td>Querying AWS CloudTrail Logs (p. 135)</td>
<td>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 automatically 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. 136).</td>
</tr>
<tr>
<td>Amazon CloudFront</td>
<td>Querying Amazon CloudFront Logs (p. 139)</td>
<td>Use Athena to query Amazon CloudFront.</td>
</tr>
<tr>
<td>Elastic Load Balancing</td>
<td>• Querying Application Load Balancer Logs (p. 143) • Querying Classic Load Balancer Logs (p. 140)</td>
<td>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. See Creating the Table for ALB Logs (p. 143) 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,</td>
</tr>
<tr>
<td>AWS Service</td>
<td>Topic</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-latency, and bytes transferred. See <a href="#">Creating the Table for ELB Logs</a>.</td>
</tr>
<tr>
<td>Amazon Virtual Private Cloud</td>
<td>Querying Amazon VPC Flow Logs (p. 145)</td>
<td>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.</td>
</tr>
<tr>
<td>AWS CloudFormation</td>
<td>AWS::Athena::NamedQuery in the AWS CloudFormation User Guide</td>
<td>Create named queries with AWS CloudFormation and run them in Athena. Named queries allow you to map a query name to a query and then call the query multiple times referencing it by its name. For information, see <a href="#">CreateNamedQuery</a> in the Amazon Athena API Reference, and AWS::Athena::NamedQuery in the AWS CloudFormation User Guide.</td>
</tr>
<tr>
<td>AWS Glue Data Catalog</td>
<td>Integration with AWS Glue (p. 28)</td>
<td>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 AWS account and integrated with the ETL and data discovery features of AWS Glue. For more information, see Integration with AWS Glue (p. 28) and What is AWS Glue in the AWS Glue Developer Guide.</td>
</tr>
<tr>
<td>AWS Service</td>
<td>Topic</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Amazon QuickSight</td>
<td>Connecting to Amazon Athena with ODBC and JDBC Drivers (p. 43)</td>
<td>Athena integrates with Amazon QuickSight for easy data visualization. You can use Athena to generate reports or to explore data with business intelligence tools or SQL clients connected with a JDBC or an ODBC driver. For more information, see What is Amazon QuickSight in the Amazon QuickSight User Guide and Connecting to Amazon Athena with ODBC and JDBC Drivers (p. 43).</td>
</tr>
<tr>
<td>IAM</td>
<td>Actions for Amazon Athena</td>
<td>You can use Athena API actions in IAM permission policies. See Actions for Amazon Athena and Access Control Policies (p. 47).</td>
</tr>
</tbody>
</table>
Release Notes

Describes Amazon Athena features, improvements, and bug fixes by release date.

Contents

- March 05, 2019 (p. 7)
- February 22, 2019 (p. 7)
- February 18, 2019 (p. 8)
- November 20, 2018 (p. 9)
- October 15, 2018 (p. 9)
- October 10, 2018 (p. 10)
- September 6, 2018 (p. 10)
- August 23, 2018 (p. 11)
- August 16, 2018 (p. 11)
- August 7, 2018 (p. 12)
- June 5, 2018 (p. 12)
  - Support for Views (p. 12)
  - Improvements and Updates to Error Messages (p. 12)
  - Bug Fixes (p. 13)
- May 17, 2018 (p. 13)
- April 19, 2018 (p. 13)
- April 6, 2018 (p. 14)
- March 15, 2018 (p. 14)
- February 2, 2018 (p. 14)
- January 19, 2018 (p. 14)
- November 13, 2017 (p. 15)
- November 1, 2017 (p. 15)
- October 19, 2017 (p. 15)
- October 3, 2017 (p. 15)
- September 25, 2017 (p. 15)
- August 14, 2017 (p. 16)
- August 4, 2017 (p. 16)
- June 22, 2017 (p. 16)
- June 8, 2017 (p. 16)
- May 19, 2017 (p. 16)
  - Improvements (p. 17)
  - Bug Fixes (p. 17)
- April 4, 2017 (p. 17)
  - Features (p. 17)
  - Improvements (p. 18)
  - Bug Fixes (p. 18)
- March 24, 2017 (p. 18)
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. 44). 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. 172) to create and manage workgroups, and Athena API tag actions (p. 182) 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. 158) and Workgroup Example Policies (p. 162).
- Tagging Workgroups (p. 179) and Tag-Based IAM Access Control Policies (p. 183).

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. 43).

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. 179).

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. 179).

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. 183).

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. 43).

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. 158). 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. 158) and Controlling Costs and Monitoring Queries with CloudWatch Metrics (p. 174).

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 IAM User Guide. Before you create new workgroups, make sure that you use workgroup IAM policies (p. 161), and the AmazonAthenaFullAccess Managed Policy (p. 48).

You can start using workgroups in the console, with the workgroup API operations (p. 172), or with the JDBC driver. For a high-level procedure, see Setting up Workgroups (p. 160). To download the JDBC driver with workgroup support, see Using Athena with the JDBC Driver (p. 43).

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. 172) and Troubleshooting Workgroups (p. 172).

- 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 the section called “OpenX JSON SerDe” (p. 203).

- 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. 68) 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 “Querying Network Load Balancer Logs” (p. 142).

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 Athena API” (p. 59).

To download the JDBC driver version 2.0.6 and its documentation, see Using Athena with the JDBC Driver (p. 43). 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. 44). 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. 58).
- 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. 51).

**Note**
Data resides in the Amazon S3 buckets, and access to it is governed by the Amazon S3 Permissions (p. 51). 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. 91).

Use CTAS queries to:

- Create a table from query results (p. 91) in one step.
- Create CTAS queries in the Athena console (p. 93), using Examples (p. 97). For information about syntax, see `CREATE TABLE AS` (p. 225).
- 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. 91) and Columnar Storage Formats (p. 78).

### 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. 44).

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. 50).

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. 216).

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](p. 5).

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](p. 5), and search for `UseResultsetStreaming`.

For downloading the JDBC driver version 2.0.5 and its documentation, see [Using Athena with the JDBC Driver](p. 43).

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. 50). 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. 248), and the JDBC Driver Migration Guide.

### August 7, 2018

Published on 2018-08-07

You can now store Amazon Virtual Private Cloud flow logs directly in Amazon S3 in a GZIP format, where you can query them in Athena. For information, see Querying Amazon VPC Flow Logs (p. 145) and Amazon VPC Flow Logs can now be delivered to S3.

### June 5, 2018

Published on 2018-06-05

**Topics**

- Support for Views (p. 12)
- Improvements and Updates to Error Messages (p. 12)
- Bug Fixes (p. 13)

**Support for Views**

Added support for views. You can now use CREATE VIEW (p. 227), DESCRIBE VIEW (p. 228), DROP VIEW (p. 229), SHOW CREATE VIEW (p. 231), and SHOW VIEWS (p. 233) in Athena. The query that defines the view runs each time you reference the view in your query. For more information, see Views (p. 86).

**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. 148).
- 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".
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 limits 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 limits are separate for DDL and SELECT queries.

Concurrency limits 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 limit, the Athena API displays an error message: "You have exceeded the limit for the number of queries you can run concurrently. Reduce the number of concurrent queries submitted by this account. Contact customer support to request a concurrent query limit increase."

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 Limits (p. 253). This is a soft limit and you can request a limit 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. 43).

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. 248).
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 Creating a Table for CloudTrail Logs in the CloudTrail Console (p. 136).

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 currently supported for queries that use the OpenCSV SerDe, 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. 233) in this guide, and Presto 0.172 Functions.

Athena does not support all of Presto's features. For more information, see Limitations (p. 238).

**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. 44).

**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. 123) 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. 28).

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. 199). 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 EU (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. 43).
• 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 Configuring Encryption Options (p. 62).

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. 248) for the previous version of the driver, and Connect with the JDBC (p. 43), 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 abort 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. 195). 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. 192)
  - OpenCSVSerDe for Processing CSV (p. 197)
- **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.
Setting Up

If you’ve already signed up for Amazon Web Services (AWS), you can start using Amazon Athena immediately. If you haven’t signed up for AWS, or if you need assistance querying data using Athena, first complete the tasks below:

Sign Up for AWS

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. When you use Athena, you use Amazon S3 to store your data. Athena has no AWS Free Tier 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.

Note your AWS account number, because you need it for the next task.

Create an IAM User

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. Create an IAM user, and then add the user to an IAM group with administrative permissions or and grant this user administrative permissions. You can then access AWS using a special URL and the credentials for the IAM user.

If you signed up for AWS but have not created an IAM user for yourself, you can create one using the IAM console. If you aren't familiar with using the console, see Working with the AWS Management Console.

To create a group for administrators

2. In the navigation pane, choose Groups, Create New Group.
3. For Group Name, type a name for your group, such as Administrators, and choose Next Step.
4. In the list of policies, select the check box next to the AdministratorAccess policy. You can use the Filter menu and the Search field to filter the list of policies.
5. Choose Next Step, Create Group. Your new group is listed under Group Name.
To create an IAM user for yourself, add the user to the administrators group, and create a password for the user

1. In the navigation pane, choose **Users**, and then **Create New Users**.
2. For 1, type a user name.
3. Clear the check box next to **Generate an access key for each user** and then **Create**.
4. In the list of users, select the name (not the check box) of the user you just created. You can use the **Search** field to search for the user name.
5. Choose **Groups, Add User to Groups**.
6. Select the check box next to the administrators and choose **Add to Groups**.
7. Choose the **Security Credentials** tab. Under **Sign-In Credentials**, choose **Manage Password**.
8. Choose **Assign a custom password**. Then type a password in the **Password** and **Confirm Password** fields. When you are finished, choose **Apply**.
9. To sign in as this new IAM user, sign out of the AWS console, then use the following URL, where `your_aws_account_id` is your AWS account number without the hyphens (for example, if your AWS account number is 1234-5678-9012, your AWS account ID is 123456789012):

   ![URL](https://*your_account_alias*.signin.aws.amazon.com/console/)

   It is also possible the sign-in link will use your account name instead of number. To verify the sign-in link for IAM users for your account, open the IAM console and check under **IAM users sign-in link** on the dashboard.

### Attach Managed Policies for Using Athena

Attach Athena managed policies to the IAM account you use to access Athena. There are two managed policies for Athena: **AmazonAthenaFullAccess** and **AWSQuicksightAthenaAccess**. These policies grant permissions to Athena to query Amazon S3 as well as write the results of your queries to a separate bucket on your behalf. For more information and step-by-step instructions, see **Attaching Managed Policies** in the **AWS Identity and Access Management User Guide**. For information about policy contents, see **IAM Policies for User Access** (p. 47).

**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. 51). If the dataset or Athena query results are encrypted, you may need additional permissions. For more information, see **Configuring Encryption Options** (p. 62).
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 is using live resources, so you are charged for the queries that you run. You aren't charged for the sample datasets that you use, 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. 21).

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, you'll go to a Getting Started page. Choose Get Started to open the Query Editor. If it isn't your first time, the Athena Query Editor opens.
3. In the Athena Query Editor, you see a query pane with an example query. Start typing your query anywhere in the query pane.

4. To create a database named mydatabase, enter the following CREATE DATABASE statement, and then choose Run Query:

   CREATE DATABASE mydatabase

5. Confirm that the catalog display refreshes and mydatabase appears in the DATABASE list in the Catalog dashboard on the left side.
Step 2: Create a Table

Now that you have a database, you're ready to create a table that's based on the sample data file. You define columns that map to the data, specify how the data is delimited, and provide the location in Amazon S3 for the file.

To create a table

1. Make sure that mydatabase is selected for DATABASE and then choose New Query.
2. In the query pane, enter the following CREATE TABLE statement, and then choose Run Query:

   Note
   You can query data in regions other than the region where you run Athena. Standard inter-region data transfer rates for Amazon S3 apply in addition to standard Athena charges. To reduce data transfer charges, replace myregion in s3://athena-examples-myregion/path/to/data/ with the region identifier where you run Athena, for example, s3://athena-examples-us-east-1/path/to/data/.

   CREATE EXTERNAL TABLE IF NOT EXISTS cloudfront_logs (  `Date` DATE,  Time STRING,  Location STRING,  Bytes INT,  RequestIP STRING,  Method STRING,  Host STRING,  Uri STRING,  Status INT,  Referrer STRING,  os STRING,  Browser STRING,  BrowserVersion STRING  ) ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe' WITH SERDEPROPERTIES ("input.regex" = "^\(?!#)([^ \]+)\s+([^ \]+)\s+([^ \]+)\s+([^ \]+)\s+([^ \]+)\s+([^ \]+)\s+([^ \]+)\s+([^ \]+)\s+([^ \]+)\s+[^\(\)]*[^\(\)]*[^\(\)]*[^\(\)]*$" ) LOCATION 's3://athena-examples-myregion/cloudfront/plaintext/';

The table cloudfront_logs is created and appears in the Catalog dashboard for your 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 queries on the table and see the results in Athena.

To run a query

1. Choose New Query, enter the following statement anywhere in the query pane, and then choose Run Query:

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

   Results are returned that look like the following:
2. Optionally, you can save the results of a query to CSV by choosing the file icon on the Results pane.

You can also view the results of previous queries or queries that may take some time to complete. Choose History then either search for your query or choose View or Download to view or download the results of previous completed queries. This also displays the status of queries that are currently running. Query history is retained for 45 days. For information, see Viewing Query History (p. 85).

Query results are also stored in Amazon S3 in a bucket called aws-athena-query-results-ACCOUNTID-REGION. You can change the default location in the console and encryption options by choosing Settings in the upper right pane. For more information, see Query Results (p. 83).
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. 240).

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.
Integration with AWS Glue

AWS Glue is a fully managed ETL (extract, transform, and load) service that can categorize your data, clean it, enrich it, and move it reliably between various data stores. AWS Glue crawlers automatically infer database and table schema from your source data, storing the associated metadata in the AWS Glue Data Catalog. When you create a table in Athena, you can choose to create it using an AWS Glue crawler.

In regions where AWS Glue is supported, Athena uses the AWS Glue Data Catalog as a central location to store and retrieve table metadata throughout an AWS account. The Athena execution engine requires table metadata that instructs it where to read data, how to read it, and other information necessary to process the data. The AWS Glue Data Catalog provides a unified metadata repository across a variety of data sources and data formats, integrating not only with Athena, but with Amazon S3, Amazon RDS, Amazon Redshift, Amazon Redshift Spectrum, Amazon EMR, and any application compatible with the Apache Hive metastore.

For more information about the AWS Glue Data Catalog, see Populating the AWS Glue Data Catalog in the AWS Glue Developer Guide. For a list of regions where AWS Glue is available, see Regions and Endpoints in the AWS General Reference.

Separate charges apply to AWS Glue. For more information, see AWS Glue Pricing and Are there separate charges for AWS Glue? (p. 32) For more information about the benefits of using AWS Glue with Athena, see Why should I upgrade to the AWS Glue Data Catalog? (p. 31)

Topics
- Upgrading to the AWS Glue Data Catalog Step-by-Step (p. 29)
- FAQ: Upgrading to the AWS Glue Data Catalog (p. 31)
- Best Practices When Using Athena with AWS Glue (p. 33)
Amazon Athena manages its own data catalog until the time that AWS Glue releases in the Athena region. At that time, if you previously created databases and tables using Athena or Amazon Redshift Spectrum, you can choose to upgrade Athena to the AWS Glue Data Catalog. If you are new to Athena, you don't need to make any changes; databases and tables are available to Athena using the AWS Glue Data Catalog and vice versa. For more information about the benefits of using the AWS Glue Data Catalog, see FAQ: Upgrading to the AWS Glue Data Catalog (p. 31). For a list of regions where AWS Glue is available, see Regions and Endpoints in the AWS General Reference.

Until you upgrade, the Athena-managed data catalog continues to store your table and database metadata, and 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, you aren't able to create tables or databases with the same names, and the creation process in either AWS Glue or Athena fails in this case.

We created a wizard in the Athena console to walk you through the steps of upgrading to the AWS Glue console. The upgrade takes just a few minutes, and you can pick up where you left off. For more information about each upgrade step, see the topics in this section. For more 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. 33).

**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 managed policy, they are automatically updated. 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. 47).
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 now have the option to create a table using an AWS Glue crawler. For more information, see Using AWS Glue Crawlers (p. 34).
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 features together with Athena and Redshift Spectrum, you must upgrade to the AWS Glue Data Catalog. Athena can only be used together with the AWS Glue Data Catalog in regions where AWS Glue is available. For a list of regions, see Regions and Endpoints in the AWS General Reference.

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, pointing to a data store in Amazon S3 or a JDBC-compliant data store.

For more information, see AWS Glue Concepts.

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

Unified metadata repository

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

**Automatic schema and partition recognition**

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

For more information, see Cataloging Tables with a Crawler.

**Easy-to-build pipelines**

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

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

**Are there separate charges for AWS Glue?**

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

**Upgrade process FAQ**

- Who can perform the upgrade? (p. 32)
- My users use a managed policy with Athena and Redshift Spectrum. What steps do I need to take to upgrade? (p. 32)
- What happens if I don’t upgrade? (p. 33)
- Why do I need to add AWS Glue policies to Athena users? (p. 33)
- What happens if I don’t allow AWS Glue policies for Athena users? (p. 33)
- Is there risk of data loss during the upgrade? (p. 33)
- Is my data also moved during this upgrade? (p. 33)

**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. 29).

**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 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 and 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, Athena actions no longer apply to accessing the AWS Glue Data Catalog, so 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. 29).

Is there risk of data loss during the upgrade?

No.

Is my data also moved during this upgrade?

No. The migration only affects metadata.

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

In this topic

- Database, Table, and Column Names (p. 34)
- Using AWS Glue Crawlers (p. 34)
  - Scheduling a Crawler to Keep the AWS Glue Data Catalog and Amazon S3 in Sync (p. 34)
  - Using Multiple Data Sources with Crawlers (p. 35)
  - Syncing Partition Schema to Avoid “HIVE_PARTITION_SCHEMA_MISMATCH” (p. 37)
  - Updating Table Metadata (p. 37)
- Working with CSV Files (p. 38)
  - CSV Data Enclosed in Quotes (p. 38)
  - CSV Files with Headers (p. 40)
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.

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 execute 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

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:

   ![Crawler info](image)

   ![Data stores](image)

   **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. 35).

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

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

   **Updating Table Metadata**

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

   AWS Glue may mis-assign metadata when a CSV file has quotes around each data field, getting the `serializationLib` property wrong. For more information, see *CSV Data Enclosed in quotes* (p. 38).
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

If you run a query in Athena against a table created from a CSV file with quoted data values, update the table definition in AWS Glue so that it specifies the right SerDe and SerDe properties. This allows the table definition to use the OpenCSV SerDe. For more information about the OpenCSV SerDe, see OpenCSVSerDe for Processing CSV (p. 197).

In this case, make the following changes:

- Change the `serializationLib` property under field in the `SerDeInfo` field in the table to `org.apache.hadoop.hive.serde2.OpenCSVSerde`.
- Enter appropriate values for `separatorChar`, `quoteChar`, and `escapeChar`. The `separatorChar` value is a comma, the `quoteChar` value is double quotes (""), and the `escapeChar` value is the backslash (\).

For example, for a CSV file with records such as the following:

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

You can use the AWS Glue console to edit table details as shown in this example:
Alternatively, you can update the table definition in AWS Glue to have a SerDeInfo block such as the following:

```
"SerDeInfo": {
"escapeChar": \\
"quoteChar": "
"separatorChar": ,
"Type key...": Type value...
```
CSV Files with Headers

If you are writing CSV files from AWS Glue to query using Athena, you must remove the CSV headers 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:

```python
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")
```

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.

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.
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. 43)
- Connecting to Amazon Athena with ODBC (p. 44)

**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.

**Links for Downloading the JDBC Driver**

The JDBC driver version 2.0.7 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 AthenaJDBC41-2.0.7.jar is compatible with JDBC 4.1 and requires JDK 7.0 or later.
- The AthenaJDBC42-2.0.7.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

Now you are ready to migrate from the previous version and install and configure this version of the JDBC driver.
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.

Migration from Previous Version of the JDBC Driver

The current JDBC driver version 2.0.7 is a drop-in replacement of the previous version of the JDBC driver version 2.0.6, and is backwards compatible with the JDBC driver version 2.0.6, with the following step that you must perform to ensure the driver runs.

**Important**

To ensure that you can use the JDBC driver versions 2.0.5 or greater, 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. 50). For more information about upgrading to versions 2.0.5 or greater from version 2.0.2, see the JDBC Driver Migration Guide. Additionally, ensure that port 444 is open to outbound traffic.

For more information about the previous versions of the JDBC driver, see Using the Previous Version of the JDBC Driver (p. 248).

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 driver version 2.x. For information, see the JDBC Driver Migration Guide.

Connecting to Amazon Athena with ODBC

Download the ODBC driver, the Amazon Athena ODBC driver License Agreement, and the documentation for the driver using the following links.

Amazon Athena ODBC Driver License Agreement

License Agreement

Windows

<table>
<thead>
<tr>
<th>Driver Version</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>
</tbody>
</table>

Linux

<table>
<thead>
<tr>
<th>Driver Version</th>
<th>Download Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC 1.0.5 for Linux 32-bit</td>
<td>Linux 32 bit ODBC Driver 1.0.5</td>
</tr>
</tbody>
</table>
### Migration from the Previous Version of the ODBC Driver

The current ODBC driver version 1.0.5 is a drop-in replacement of the previous version of the ODBC driver version 1.0.4. 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 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 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. 50).

### Previous Versions of the ODBC Driver

<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>

### OSX

<table>
<thead>
<tr>
<th>Driver Version 1.0.5</th>
<th>Download Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC 1.0.5 for OSX</td>
<td>OSX ODBC Driver 1.0.5</td>
</tr>
</tbody>
</table>

### ODBC Driver Documentation

<table>
<thead>
<tr>
<th>Driver Version</th>
<th>Download Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documentation for ODBC 1.0.5</td>
<td>ODBC Driver Installation and Configuration Guide version 1.0.5</td>
</tr>
<tr>
<td>Release Notes for ODBC 1.0.5</td>
<td>ODBC Driver Release Notes version 1.0.5</td>
</tr>
</tbody>
</table>
### Previous Versions of the ODBC Driver

<table>
<thead>
<tr>
<th>Driver Version 1.0.4</th>
<th>Download Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documentation for ODBC 1.0.4</td>
<td>ODBC Driver Installation and Configuration Guide version 1.0.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Driver Version 1.0.3</th>
<th>Download Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC 1.0.3 for Windows 32-bit</td>
<td>Windows 32-bit ODBC Driver 1.0.3</td>
</tr>
<tr>
<td>ODBC 1.0.3 for Windows 64-bit</td>
<td>Windows 64-bit ODBC Driver 1.0.3</td>
</tr>
<tr>
<td>ODBC 1.0.3 for Linux 32-bit</td>
<td>Linux 32-bit ODBC Driver 1.0.3</td>
</tr>
<tr>
<td>ODBC 1.0.3 for Linux 64-bit</td>
<td>Linux 64-bit ODBC Driver 1.0.3</td>
</tr>
<tr>
<td>ODBC 1.0.3 for OSX</td>
<td>OSX ODBC Driver 1.0</td>
</tr>
<tr>
<td>Documentation for ODBC 1.0.3</td>
<td>ODBC Driver Installation and Configuration Guide version 1.0.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Driver Version 1.0.2</th>
<th>Download Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC 1.0.2 for Windows 32-bit</td>
<td>Windows 32-bit ODBC Driver 1.0.2</td>
</tr>
<tr>
<td>ODBC 1.0.2 for Windows 64-bit</td>
<td>Windows 64-bit ODBC Driver 1.0.2</td>
</tr>
<tr>
<td>ODBC 1.0.2 for Linux 32-bit</td>
<td>Linux 32-bit ODBC Driver 1.0.2</td>
</tr>
<tr>
<td>ODBC 1.0.2 for Linux 64-bit</td>
<td>Linux 64-bit ODBC Driver 1.0.2</td>
</tr>
<tr>
<td>ODBC 1.0 for OSX</td>
<td>OSX ODBC Driver 1.0</td>
</tr>
<tr>
<td>Documentation for ODBC 1.0.2</td>
<td>ODBC Driver Installation and Configuration Guide version 1.0.2</td>
</tr>
</tbody>
</table>
Security

Amazon Athena uses IAM policies to restrict access to Athena operations. Encryption options enable you to encrypt query result files in Amazon S3 and query data encrypted in Amazon S3. Users must have the appropriate permissions to access the Amazon S3 locations and decrypt files.

Topics
- Access Control Policies (p. 47)
- Configuring Encryption Options (p. 62)

Access Control Policies

To run queries in Athena, you must have the appropriate permissions for:

- The Athena actions.
- The Amazon S3 locations where the underlying data is stored that you are going to query in Athena.
- The resources that you store in AWS Glue Data Catalog, such as databases and tables, that you are going to query in Athena.
- The encrypted metadata in the AWS Glue Data Catalog (if you migrated to using that metadata in Athena and the metadata is encrypted).
- Actions on Athena workgroups (p. 158).

If you are an administrator for other users, make sure that they have appropriate permissions associated with their user profiles.

Topics
- Managed Policies for User Access (p. 47)
- Access to Amazon S3 (p. 51)
- Fine-Grained Access to Databases and Tables in the AWS Glue Data Catalog (p. 51)
- Access to Encrypted Metadata in the AWS Glue Data Catalog (p. 58)
- Cross-account Access (p. 58)
- Workgroup and Tag Policies (p. 59)
- Enabling Federated Access to Athena API (p. 59)

Managed Policies for User Access

To allow or deny Athena service actions for yourself or other users, use identity-based (IAM) policies attached to principals, such as users or groups.

Each identity-based (IAM) policy consists of statements that define the actions that are allowed or denied. For a list of actions, see the Amazon Athena API Reference.

Managed policies are easy to use and are automatically updated with the required actions as the service evolves. For more information and step-by-step instructions for attaching a policy to a user, see Attaching Managed Policies in the AWS Identity and Access Management User Guide.

Athena has these managed policies:
• The AmazonAthenaFullAccess managed policy grants full access to Athena. Attach it to users and other principals who need full access to Athena. See AmazonAthenaFullAccess Managed Policy (p. 48).

• The AWSQuicksightAthenaAccess managed policy grants access to actions that Amazon QuickSight needs to integrate with Athena. Attach this policy to principals who use Amazon QuickSight in conjunction with Athena. See AWSQuicksightAthenaAccess Managed Policy (p. 50).

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 AWS Identity and Access Management 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. 51).

If you have migrated to using AWS Glue with Athena, and have chosen to encrypt your AWS Glue Data Catalog, you must specify additional actions to the identity-based IAM policies that you use in Athena. For more information, see Access to Encrypted Metadata in the AWS Glue Data Catalog (p. 58).

Important
If you create and use workgroups, make sure your policies include appropriate access to workgroup actions. For detailed information, see the section called “IAM Policies for Accessing Workgroups” (p. 161) and the section called “Workgroup Example Policies” (p. 162).

AmazonAthenaFullAccess Managed Policy

The AmazonAthenaFullAccess managed policy grants full access to Athena.

Managed policy contents change, so the policy shown here may be out-of-date. Check the IAM console for the most up-to-date policy.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "athena:*"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "glue:CreateDatabase",
        "glue:DeleteDatabase",
        "glue:GetDatabase",
        "glue:GetDatabases",
        "glue:UpdateDatabase",
        "glue:CreateTable",
        "glue:DeleteTable",
        "glue:BatchDeleteTable",
        "glue: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"
],
"Resource": [
  "arn:aws:s3:::aws-athena-query-results-*"
]
},
{
"Effect": "Allow",
"Action": [
  "s3:GetObject",
  "s3:ListBucket"
],
"Resource": [
  "arn:aws:s3:::athena-examples*"
]
},
{
"Effect": "Allow",
"Action": [
  "s3:ListBucket",
  "s3:GetBucketLocation",
  "s3:ListAllMyBuckets"
],
"Resource": [*
]
},
{
"Effect": "Allow",
"Action": [
  "sns:ListTopics",
  "sns:GetTopicAttributes"
],
"Resource": [*
]
},
{
"Effect": "Allow",
"Action": [
  "cloudwatch:PutMetricAlarm",
  "cloudwatch:DescribeAlarms",
  "cloudwatch:DeleteAlarms"
],
"Resource": [*
]
}
AWSQuicksightAthenaAccess Managed Policy

An additional managed policy, AWSQuicksightAthenaAccess, grants access to actions that Amazon QuickSight needs to integrate 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. Attach this policy only to principals who use Amazon QuickSight in conjunction with Athena.

Managed policy contents change, so the policy shown here may be out-of-date. Check the IAM console for the most up-to-date policy.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "athena:BatchGetQueryExecution",
        "athena:CancelQueryExecution",
        "athena:GetCatalogs",
        "athena:GetExecutionEngine",
        "athena:GetExecutionEngines",
        "athena:GetNamespace",
        "athena:GetNamespaces",
        "athena:GetQueryExecution",
        "athena:GetQueryExecutions",
        "athena:GetQueryResults",
        "athena:GetQueryResultsStream",
        "athena:GetTable",
        "athena:GetTables",
        "athena:ListQueryExecutions",
        "athena:RunQuery",
        "athena:StartQueryExecution",
        "athena:StopQueryExecution"
      ],
      "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",
```
Access to Amazon S3

In addition to the allowed actions for Athena that you define in IAM identity-based policies, if you or your users need to create tables and work with underlying data, you must grant appropriate access to the Amazon S3 location of the data.

You can do this using identity-based policies, bucket resource policies, or both. 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 later in Cross-Account Access (p. 58).

**Note**

Athena does not support restricting or allowing access to Amazon S3 resources based on the aws:SourceIp condition key.

Fine-Grained Access to Databases and Tables in the AWS Glue Data Catalog

After you upgrade (p. 29) to the AWS Glue Data Catalog, you can define resource-level policies for the following Data Catalog objects that are used in Athena:
These policies enable you to define fine-grained access to databases and tables. You define resource-level permissions in identity-based (IAM) policies in the IAM Console.

**Important**

This section discusses identity-based (IAM) policies that allow you to define fine-grained access to specific resources. These are not the same as resource-based policies. For more information about the differences, see Identity-Based Policies and Resource-Based Policies in the AWS Identity and Access Management 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 AWS Identity and Access Management User Guide.</td>
</tr>
<tr>
<td>Learn about identity-based (IAM) 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. 52)
- Mandatory: Access Policy to the Default Database and Catalog per AWS Region (p. 53)
- Table Partitions and Versions in AWS Glue (p. 54)
- Fine-Grained Policy Examples (p. 54)

**Limitations**

Consider the following limitations when using fine-grained access control with the AWS Glue Data Catalog and Athena:

- You must **upgrade** (p. 29) from a data catalog managed in Athena to the AWS Glue Data Catalog.
- 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. 54).
- Athena does not support cross-account access to the AWS Glue Data Catalog.
- 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. Functions are specific to AWS Glue, however, for delete actions in Athena, you must include permissions to them, if you delete a database. See Fine-Grained Policy Examples (p. 54).

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 this resource
(table or database) and all ancestors of this resource in the Data Catalog. For example, for a table, its
ancestors are database and catalog for the account. For a database, its ancestor is the catalog for this
account. See Fine-Grained Policy Examples (p. 54).

- For a delete action in Athena, such as drop database or drop table, you additionally need
permissions to call this delete action on all descendants of this resource in the Data Catalog. For
example, to delete a database you need permissions on the database, on its ancestor, which is the
catalog, and on all descendants, which are the tables and the user-defined functions. A table does not
have descendants, and therefore, to run drop table, you need permissions to this action on the table
and its ancestors. See Fine-Grained Policy Examples (p. 54).

- When limiting access to a specific database in the Data Catalog, you must also specify the access policy
to the default database and catalog for each AWS Region for get database and create database
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 get database 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:
get database and create database:

```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**

The following access policy to the default database and to the AWS Glue Data Catalog per AWS Region
for get database and create database must be present for Athena to work with the AWS Glue Data
Catalog:

```json
{
  "Effect": "Allow",
  "Action": [
    "glue:GetDatabase",
    "glue:CreateDatabase"
  ],
  "Resource": [
    "arn:aws:glue:us-east-1:123456789012:catalog",
    "arn:aws:glue:us-east-1:123456789012:database/default"
  ]
}
```
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, this means that:

- 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/acl.html) in the *AWS Glue Developer Guide*.

Examples of Fine-Grained Permissions to Tables and Databases

The following table lists examples of identity-based (IAM) policies that allow fine-grained access to databases and tables in Athena.

As with any IAM policy, you define these policies in the [IAM Console](https://console.aws.amazon.com/iam/). 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>
<tr>
<td>DDL Statement</td>
<td>Example of an IAM access policy granting access to the resource</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>ALTER DATABASE</strong></td>
<td>Allows you to modify the properties for the example_db database.</td>
</tr>
</tbody>
</table>
| | ```
| { |
|   "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:UpdateDatabase"
|   ],
|   "Resource": [ |
|     "arn:aws:glue:us-east-1:123456789012:catalog",
|     "arn:aws:glue:us-east-1:123456789012:database/example_db"
|   ]
| }
| **DROP DATABASE** | Allows you to drop the example_db database, including all tables in it. |
| | ```
| { |
|   "Effect": "Allow",
|   "Action": [ |
|     "glue:GetDatabase",
|     "glue:CreateDatabase"
|   ],
|   "Resource": [ |
|     "arn:aws:glue:us-east-1:123456789012:catalog",
|     "arn:aws:glue:us-east-1:123456789012:database/default"
|   ]
| },
| { |
|   "Effect": "Allow",
|   "Action": [ |
|     "glue:GetDatabase",
|     "glue:DeleteDatabase",
|     "glue:GetTables",
|     "glue:GetTable",
|     "glue:DeleteTable"
|   ],
|   "Resource": [ |
|     "arn:aws:glue:us-east-1:123456789012:catalog",
|     "arn:aws:glue:us-east-1:123456789012:database/example_db",
|     "arn:aws:glue:us-east-1:123456789012:table/example_db/*",
|     "arn:aws:glue:us-east-1:123456789012:userDefinedFunction/example_db/*"
|   ]
| }
|  ```
<table>
<thead>
<tr>
<th>DDL Statement</th>
<th>Example of an IAM access policy granting access to the resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHOW DATABASES</td>
<td>Allows you to list all databases in the AWS Glue Data Catalog.</td>
</tr>
</tbody>
</table>

```json
{
  "Effect": "Allow",
  "Action": [
    "glue:GetDatabase",
    "glue:CreateDatabase"
  ],
  "Resource": [
    "arn:aws:glue:us-east-1:123456789012:catalog",
    "arn:aws:glue:us-east-1:123456789012:database/default"
  ],
}
{
  "Effect": "Allow",
  "Action": [
    "glue:GetDatabases"
  ],
  "Resource": [
    "arn:aws:glue:us-east-1:123456789012:catalog",
    "arn:aws:glue:us-east-1:123456789012:database/*"
  ]
}
```

<table>
<thead>
<tr>
<th>CREATE TABLE</th>
<th>Allows you to create a table named test in the example_db database.</th>
</tr>
</thead>
</table>

```json
{
  "Effect": "Allow",
  "Action": [
    "glue:GetDatabase",
    "glue:CreateDatabase"
  ],
  "Resource": [
    "arn:aws:glue:us-east-1:123456789012:catalog",
    "arn:aws:glue:us-east-1:123456789012:database/default"
  ],
}
{
  "Effect": "Allow",
  "Action": [
    "glue:GetDatabase",
    "glue:GetTable",
    "glue:CreateTable"
  ],
  "Resource": [
    "arn:aws:glue:us-east-1:123456789012:catalog",
    "arn:aws:glue:us-east-1:123456789012:database/example_db",
    "arn:aws:glue:us-east-1:123456789012:table/example_db/test"
  ]
}
```
<table>
<thead>
<tr>
<th>DDL Statement</th>
<th>Example of an IAM access policy granting access to the resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHOW TABLES</td>
<td>Allows you to list all tables in the example_db database.</td>
</tr>
</tbody>
</table>
|               | {
|               |   "Effect": "Allow",
|               |   "Action": ["glue:GetDatabase","glue:CreateDatabase"],
|               |     "arn:aws:glue:us-east-1:123456789012:database/default"]
|               | }, {
|               |   "Effect": "Allow",
|               |   "Action": ["glue:GetDatabase","glue:GetTables"],
|               |     "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"],
|               |     "arn:aws:glue:us-east-1:123456789012:database/default"]
|               | }, {
|               |   "Effect": "Allow",
|               |   "Action": ["glue:GetDatabase","glue:GetTable","glue:DeleteTable","glue:GetPartitions","glue:GetPartition","glue:DeletePartition"],
|               |     "arn:aws:glue:us-east-1:123456789012:database/example_db",
|               |     "arn:aws:glue:us-east-1:123456789012:table/example_db/test"]
|               | } |
Access to Encrypted Metadata in the AWS Glue Data Catalog

The encryption of objects in the AWS Glue Data Catalog is available only if you have upgraded to using AWS Glue with Athena.

You can optionally 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 you encrypt your AWS Glue Data Catalog, you must add the following actions to all of your policies used to access Athena:

```json
{
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
    "Action": ["kms:GenerateDataKey", "kms:Decrypt", "kms:Encrypt"],
    "Resource": "(arn of key being used to encrypt the catalog)"
  }
}
```

Cross-account Access

A common 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.

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"
      },
    }
  ]
}
```
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.

Workgroup and Tag Policies

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 workgroup named test_workgroup, specify it as a resource as follows:

"Resource": ["arn:aws:athena:us-east-1:123456789012:workgroup/test_workgroup"]

For a list of workgroup policies, see the section called "Workgroup Example Policies" (p. 162).

For a list of tag-based policies for workgroups, see Tag-Based IAM Access Control Policies (p. 183).

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. 161).

Enabling Federated Access to Athena API

This section discusses federated access that allows a user or client application in your organization to call Athena API operations. In this case, your organization’s users don’t have direct access to Athena. Instead, you manage user credentials outside of AWS in Microsoft Active Directory. Active Directory supports SAML 2.0 (Security Assertion Markup Language 2.0).

In this case, to authenticate users, use the JDBC or ODBC driver with SAML.2.0 support that accesses Active Directory Federation Services (AD FS) 3.0 and enables a client application to call Athena API operations.

For more information about SAML 2.0 support in AWS, see About SAML 2.0 Federation in the IAM User Guide.

Note
Federated access to Athena API is supported for a particular type of identity provider (IdP), the Active Directory Federation Service (AD FS 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. 43) and Connecting to Amazon Athena with ODBC (p. 44).

Topics
- Before You Begin (p. 60)
- Architecture Diagram (p. 60)
- Procedure: SAML-based Federated Access to Athena API (p. 61)
Before You Begin

Before you begin, complete the following prerequisites:

- Inside your organization, install and configure the AD FS 3.0 as your IdP.
- On the Athena side, install and configure the latest available versions of JDBC or ODBC drivers that include support for federated access compatible with SAML 2.0. For information, see Using Athena with the JDBC Driver (p. 43) and Connecting to Amazon Athena with ODBC (p. 44).

Architecture Diagram

The following diagram illustrates this process.

In this diagram:

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 AD FS 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 Athena API

In this procedure, actions take place either on the side of your organization, or within AWS. The procedure provides steps that establish trust between your organization's IdP and your AWS account to enable SAML-based federated access to the AWS 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 (AD FS), 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 (AD FS) 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:
   a. In the role's permission policy, list actions that users from your organization are allowed to do in AWS.
   b. 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 (AD FS), 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 AD FS, see the blog post: Enabling Federation to AWS Using Windows Active Directory, AD FS, 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. 43) and Connecting to Amazon Athena with ODBC (p. 44).

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.

The high-level summary of configuring the connection string to the drivers is as follows:

- 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 AD FS IdP.
- For idp_host, provide the host name of the AD FS IdP server.
- For idp_port, provide the port number that the AD FS IdP listens on for the SAML assertion request.
• 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.

• Optionally, set ssl_insecure to true. In this case, the driver does not check the authenticity of the SSL certificate for the AD FS IdP server. Setting to true is needed if the AD FS IdP's SSL certificate has not been configured to be trusted by the driver.

• 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, 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. 60).

2. AWS makes sure that the request to assume the role comes from the IdP referenced in the SAML provider entity.

3. If the request is successful, the AWS STS AssumeRoleWithSAML API operation returns a set of temporary security credentials, which your client application uses to make signed requests to Athena.

Your application now has information about the current user and can access Athena programmatically.

**Configuring Encryption Options**

You can run queries in Athena to query encrypted data in Amazon S3 by indicating data encryption when you create a table. You can also encrypt the query results in Amazon S3, and the data in the AWS Glue Data Catalog.

You can encrypt:

• The results of all queries in Amazon S3, which Athena stores in a location known as the S3 staging directory. You can encrypt query results stored in Amazon S3 whether the underlying dataset is encrypted in Amazon S3 or not. For information, see Permissions to Encrypted Query Results Stored in Amazon S3 (p. 63).

• The data in the AWS Glue Data Catalog. For information, see Permissions to Encrypted Metadata in the AWS Glue Data Catalog (p. 65).

**Topics**

- Amazon S3 Encryption Options Supported in Athena (p. 62)
- Encrypting Query Results Stored in Amazon S3 (p. 63)
- Permissions to Encrypted Data in Amazon S3 (p. 64)
- Permissions to Encrypted Metadata in the AWS Glue Data Catalog (p. 65)
- Creating Tables Based on Encrypted Datasets in Amazon S3 (p. 65)

**Amazon S3 Encryption Options Supported in Athena**

Athena supports the following Amazon S3 encryption options, both for encrypted datasets in Amazon S3 and for encrypted query results:
• Server side encryption (SSE) with an Amazon S3-managed key (SSE-S3)
• Server-side encryption (SSE) with a AWS Key Management Service customer managed key (SSE-KMS).
• Client-side encryption (CSE) with a AWS KMS customer managed key (CSE-KMS)

**Note**
With SSE-KMS, Athena does not require you to indicate that data is encrypted when creating a table.

These options encrypt data at rest in Amazon S3. Regardless of whether you use these options, transport layer security (TLS) encrypts objects in-transit between Athena resources and between Athena and Amazon S3. Query results stream to JDBC clients as plain text and are encrypted using TLS.

**Important**
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.

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.

Athena does not support SSE with customer-provided keys (SSE-C), nor does it support client-side encryption using a client-side master key. To compare Amazon S3 encryption options, see Protecting Data Using Encryption in the Amazon Simple Storage Service Developer Guide.

Athena does not support running queries from one Region on encrypted data stored in Amazon S3 in another Region.

## Encrypting Query Results Stored in Amazon S3

You set up query-result encryption using the Athena console. If you connect using the JDBC driver, you configure driver options. Specify the type of encryption to use and the Amazon S3 staging directory location.

To configure the JDBC driver to encrypt your query results using any of the encryption protocols that Athena supports, see Using Athena with the JDBC Driver (p. 43).

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. 167) 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. 166). 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 driver are not used. For more information, see Workgroup Settings Override Client-Side Settings (p. 166).
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*.

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

7. Choose **Save**.

### 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. 47)** and **Amazon S3 Permissions (p. 51)**.

- **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 keys (CMKs) that are used to encrypt data in Amazon S3. The easiest way to do this is to use the IAM console to add key users to the appropriate AWS KMS key policies. For information about how to add a user to a AWS KMS key policy, see **How to Modify a Key Policy** 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, visit the AWS Support Center and create a case to increase your limit. For more information about limits and AWS KMS throttling, see Limits in the AWS Key Management Service Developer Guide.

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. 58).

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 the proper materials to use 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 appropriate permissions as 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.

Indicate that the dataset is encrypted in Amazon S3 in one of the following ways. This step is not required if SSE-KMS is used.

1. Use the CREATE TABLE (p. 222) statement with a `TBLPROPERTIES` clause that specifies `'has_encrypted_data'='true'.`

2. Use the JDBC driver (p. 43) and set the `TBLPROPERTIES` value as shown in the previous example, when you execute CREATE TABLE (p. 222) using `statement.executeQuery()`.  

65
- Use the *Add table* wizard in the Athena console, and then choose *Encrypted data set* when you specify a value for *Location of input data set*.

Tables based on encrypted data in Amazon S3 appear in the *Database* list with an encryption icon.
Working with Source Data

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 read-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

• Tables and Databases Creation Process in Athena (p. 67)
• Names for Tables, Databases, and Columns (p. 71)
• Reserved Keywords (p. 72)
• Table Location in Amazon S3 (p. 73)
• Partitioning Data (p. 74)
• Columnar Storage Formats (p. 78)
• Converting to Columnar Formats (p. 78)

Tables and Databases Creation Process in Athena

You can run DDL statements in the Athena console, using a JDBC or an ODBC driver, or using the Athena Create Table wizard.

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 execute 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 Data Formats, SerDes, and Compression Formats (p. 191).

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 Setting User and Amazon S3 Bucket Permissions (p. 47).
- If the data is not encrypted in Amazon S3, it can be stored in a different Region from the primary region where you run Athena. Standard inter-region data transfer rates for Amazon S3 apply in addition to standard Athena charges.
- If the data is encrypted in Amazon S3, it must be stored in the same Region, and the user or principal who creates the table in Athena must have the appropriate permissions to decrypt the data. For more information, see Configuring Encryption Options (p. 62).
- 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 does not support Requester Pays buckets.
- Athena does not support querying the data in the GLACIER storage class. It ignores objects transitioned to the GLACIER storage class based on an Amazon S3 lifecycle policy.

For more information, 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 Presto 0.172 Functions and Operators in the Presto documentation.

Transactional Data Transformations Are Not Supported

Athena does not support transaction-based operations (such as the ones found in Hive or Presto) on table data. For a full list of keywords not supported, see Unsupported DDL (p. 237).

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.

UDF and UDAF Are Not Supported

User-defined functions (UDF or UDAFs) and stored procedures are not supported.

To create a table using the AWS Glue Data Catalog

2. Choose AWS Glue Data Catalog. You can now create a table with the AWS Glue Crawler. For more information, see Using AWS Glue Crawlers (p. 34).

To create a table using the wizard

2. Under the database display in the Query Editor, choose Add table, which displays a wizard.
3. Follow the steps for creating your table.

To create a database using Hive DDL

A database in Athena is a logical grouping for tables you create in it.

2. Choose Query Editor.
3. Enter `CREATE DATABASE myDatabase` and choose Run Query.
4. Select your database from the menu. It is likely to be an empty database.

To create a table using Hive DDL

The Athena Query Editor displays the current database. If you create a table and don't specify a database, the table is created in the database chosen in the Databases section on the Catalog tab.

1. In the database that you created, create a table by entering the following statement and choosing Run Query:

```sql
CREATE EXTERNAL TABLE IF NOT EXISTS cloudfront_logs (  
`Date` Date,  
Time STRING,  
Location STRING,  
Bytes INT,  
RequestIP STRING,
```
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 lowercase, but Spark requires lowercase table and column names.

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

Athena table, view, database, and column names allow only underscore special characters

Athena table, view, database, and column names cannot contain special characters, other than underscore (_).

Names that begin with an underscore

Use backtics to enclose table, view, or column names that begin with an underscore. For example:

```
CREATE TABLE `_myunderscoretable` ( 
`_id` string, 
`_index`string, 
...
```

Table or view names that include numbers

Enclose table names that include numbers in quotation marks. For example:

```
CREATE TABLE "table123" 
`_id` string, 
`_index` string,
```
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. 86), enclose them in double quotes (").

- List of Reserved Keywords in DDL Statements (p. 72)
- List of Reserved Keywords in SQL SELECT Statements (p. 72)
- Examples of Queries with Reserved Keywords (p. 73)

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 (´).

```plaintext
ALL, ALTER, AND, ARRAY, AS, AUTHORIZATION, BETWEEN, BIGINT, BINARY, BOOLEAN, BOTH, BY, CASE, CAST, CHAR, COLUMN, CONF, CONSTRAINT, COMMIT, CREATE, CROSS, CUBE, CURRENT, CURRENT_DATE, CURRENT_TIMESTAMP, CURSOR, DATABASE, DATE, DAYOFWEEK, DECIMAL, DELETE, DESCRIBE, DISTINCT, DOUBLE, DROP, ELSE, END, EXCHANGE, EXISTS, EXTENDED, EXTERNAL, EXTRACT, FALSE, FETCH, FLOAT, FLOOR, FOLLOWING, FOR, FOREIGN, FROM, FULL, FUNCTION, GRANT, GROUP, GROUPING, HAVING, IF, IMPORT, IN, INNER, INSERT, INT, INTEGER, INTERSECT, INTERVAL, INTO, IS, JOIN, LATERAL, LEFT, LESS, LIKE, LOCAL, MACRO, MAP, MORE, NONE, NOT, NULL, NUMERIC, OF, ON, ONLY, OR, ORDER, OUTER, OVER, PARTIALSCAN, PARTITION, PRECENT, PRECEDING, PRECISION, PRESERVE, PRIMARY, PROCEDURE, RANGE, READS, REDUCE, REGEXP, REFERENCES, REVOKE, RIGHT, RLIKE, ROLLBACK, ROLLUP, ROW, ROWS, SELECT, SET, SMALLINT, START, TABLE, TABLESAMPLE, THEN, TIME, TIMESTAMP, TO, TRANSFORM, TRIGGER, TRUE, TRUNCATE, UNBOUNDED, UNION, UNIQUEJOIN, UPDATE, USER, USING, UTC_TIMESTAMP, VALUES, VARCHAR, VIEWS, WHEN, WHERE, WINDOW, WITH
```

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.

```plaintext
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, EXECUTS, 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, ON, OR, ORDER, OUTER, PREPARE,
```
Examples of Queries with Reserved Words

The query in the following example uses backticks (``) to escape the DDL-related reserved keywords `partition` and `date` that are used for a table name and one of the column names:

```sql
CREATE EXTERNAL TABLE `partition` ( `date` INT, col2 STRING ) PARTITIONED BY (year STRING) STORED AS TEXTFILE LOCATION 's3://test_bucket/test_examples/';
```

The following example queries include a column name containing the DDL-related reserved keywords in `ALTER TABLE ADD PARTITION` and `ALTER TABLE DROP PARTITION` statements. The DDL reserved keywords are enclosed in backticks (``):

```sql
ALTER TABLE test_table
ADD PARTITION (``date`` = '2018-05-14')

ALTER TABLE test_table
DROP PARTITION (``partition`` = 'test_partition_value')
```

The following example query includes a reserved keyword (end) as an identifier in a `SELECT` statement. The keyword is escaped in double quotes:

```sql
SELECT * FROM TestTable WHERE "end" != nil;
```

The following example query includes a reserved keyword (first) in a `SELECT` statement. The keyword is escaped in double quotes:

```sql
SELECT "itemId"."first" FROM testTable LIMIT 10;
```

Table Location in Amazon S3

When you run a `CREATE TABLE` query in Athena, you register your table with the data catalog that Athena uses. If you migrated to AWS Glue, this is the catalog from AWS Glue. You also specify the location in Amazon S3 for your table in this format: `s3://bucketname/keyname/`.

Use these tips and examples when you specify the location in Amazon S3.

- Athena reads all files in an Amazon S3 location you specify in the `CREATE TABLE` statement, and cannot ignore any files included in the prefix. When you create tables, include in the Amazon S3 path only the files you want Athena to read. Use AWS Lambda functions to scan files in the source location, remove any empty files, and move unneeded files to another location.
- In the `LOCATION` clause, use a trailing slash for your bucket.
Partitioning Data

By partitioning your data, you can restrict the amount of data scanned by each query, thus improving performance and reducing cost. Athena leverages 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.

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.

To create a table with partitions, you must define it during the CREATE TABLE statement. Use PARTITIONED BY to define the keys by which to partition data. There are two scenarios discussed below:

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

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'
  with serdeproperties ( 'paths'='requestBeginTime, adId, impressionId, referrer, userAgent, userCookie, ip' )
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 Data Formats, SerDes, and Compression Formats (p. 191).

After you execute this statement in Athena, choose New Query and execute:

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:
Scenario 2: Data is not partitioned

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/elb/plaintext/ --recursive
```

```
2016-11-23 17:54:46     11789573 elb/plaintext/2015/01/01/part-r-00000-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:46     8776899 elb/plaintext/2015/01/01/part-r-00001-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:46     9309800 elb/plaintext/2015/01/01/part-r-00002-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:47     9412570 elb/plaintext/2015/01/01/part-r-00003-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
2016-11-23 17:54:47    10725938 elb/plaintext/2015/01/01/part-r-00004-ce65fca5-d6c6-40e6-b1f9-190cc4f93814.txt
```

Scenario 2: Data is not partitioned

This query should show you data similar to the following:

```
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
```
In this case, you would have to use `ALTER TABLE ADD PARTITION` to add each partition manually.

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

```
ALTER TABLE elb_logs_raw_native_part ADD PARTITION (year='2015', month='01', day='01')
location 's3://athena-examples/elb/plaintext/2015/01/01/'
```

You can also automate adding partitions by using the JDBC driver (p. 43).
Columnar Storage Formats

Apache Parquet and ORC are columnar storage formats that are optimized for fast retrieval of data and used in AWS analytical applications.

Columnar storage formats have the following characteristics that make them suitable for using with Athena:

- **Compression by column**, with compression algorithm selected for the column data type to save storage space in Amazon S3 and reduce disk space and I/O during query processing.

- **Predicate pushdown** in Parquet and ORC enables Athena queries to fetch only the blocks it needs, improving query performance. When an Athena query obtains specific column values from your data, it uses statistics from data block predicates, such as max/min values, to determine whether to read or skip the block.

- **Splitting of data** in Parquet and ORC allows Athena to split the reading of data to multiple readers and increase parallelism during its query processing.

To convert your existing raw data from other storage formats to Parquet or ORC, you can run **CREATE TABLE AS SELECT (CTAS)** (p. 91) queries in Athena and specify a data storage format as Parquet or ORC, or use the AWS Glue Crawler.

Converting to Columnar Formats

Your Amazon Athena query performance improves if you convert your data into open source columnar formats, such as Apache Parquet or ORC.

**Note**

Use the **CREATE TABLE AS (CTAS)** (p. 98) 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/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://${REGION}.elasticmapreduce/samples/hive-ads/tables/impressions/';

Note

Replace REGION in the LOCATION clause with the region where you are running queries. For example, if your console is in us-east-1, REGION is s3://us-east-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": "TRktxshQXAHWo261jAHubijAoNlAgA",
  "browserCookie": "mvlrdwrmev",
  "userCookie": "emFlrLGrm5fa2XlFT5npwbPuG7kf6X",
  "requestEndTime": "1239714001000",
  "impressionId": "1I5G20RmOuG2rt7fFGsaWk9Xpkfb",
  "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; InfoPa",
  "timers": {
    "modelLookup": "0.3292",
    "requestTime": "0.6398"
  },
  "threadId": "99",
  "ip": "67.189.155.225",
  "modelId": "bxxiuxduaad",
  "hostname": "ec2-0-51-75-39.amazon.com",
  "sessionId": "J9H0ccA3dDMPlixCuSotl9Q8bjs6a5",
  "requestBeginTime": "1239714000000"
}
```

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

```sql
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,
useragent,
usercookie,
ip
FROM impressions WHERE dt='2009-04-14-04-05';
```

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](https://docs.aws.amazon.com/emr/latest/management-guide/).  
- Follow the instructions found in [Setting Up](p. 21).

### 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](https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-install.html) 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-east-1
   export SAMPLEURI=s3://{{REGION}}.elasticmapreduce/samples/hive-ads/tables/impressions/
   export S3BUCKET=myBucketName
   ```
Example: Converting data to Parquet using an EMR cluster

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=CONTINUE,
  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:

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

<table>
<thead>
<tr>
<th>RequestReference</th>
<th>Action</th>
<th>ImpressionID</th>
<th>UserAgent</th>
<th>Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234567890123</td>
<td>add</td>
<td></td>
<td>cartonetwork.com</td>
<td>True</td>
</tr>
<tr>
<td>2345678901234</td>
<td>update</td>
<td></td>
<td>cartonetwork.com</td>
<td>True</td>
</tr>
<tr>
<td>3456789012345</td>
<td>delete</td>
<td></td>
<td>cartonetwork.com</td>
<td>True</td>
</tr>
</tbody>
</table>

**Note:** This example demonstrates how data can be converted to the Parquet format using Amazon EMR.
Querying Data in Amazon Athena Tables

Examples of Athena queries in this section show you how to work with arrays, concatenate, filter, flatten, sort, and query data in them. Other examples include queries for data in tables with nested structures and maps, and tables that contain JSON-encoded values.

Topics
- Query Results (p. 83)
- Viewing Query History (p. 85)
- Views (p. 86)
- Creating a Table from Query Results (CTAS) (p. 91)
- Querying Arrays (p. 100)
- Querying Arrays with Complex Types and Nested Structures (p. 109)
- Querying Arrays with Maps (p. 115)
- Querying JSON (p. 116)

Query Results

Athena stores query results in Amazon S3.

Each query that you run has:

- A results file stored automatically in a CSV format (*.csv), and
- An Athena metadata file (*.csv.metadata).

**Note**
You can delete metadata files (*.csv.metadata) without causing errors, but important information about the query is lost.

If necessary, you can access the result files to work with them.

Topics
- Saving Query Results (p. 84)
- The Default Location for Query Results (p. 84)

You can specify the query results location in one of the following ways:

- **For individual queries.** This way of specifying the settings is known as *client-side query settings*. The client-side settings include query results location and encryption configuration. To view or change the query results location for individual queries, choose **Settings** in the upper right pane of the Athena console, or use the **OutputLocation** API.

- **For all queries in a workgroup.** This way of specifying the settings is known as *workgroup settings*. The workgroup settings include the results location and encryption configuration. To view or change the query results location for all queries in a workgroup, choose the **Workgroup:**<workgroup_name> tab in the console, switch to your workgroup, view or edit the workgroup, and specify the location in the **Query result location** field. For more information, see Workgroup Settings (p. 165) and Managing Workgroups (p. 167). You can also specify this setting with **WorkGroupConfiguration** in the API.
Important
If you use the API or the drivers, you must specify the query results location in one of the two ways: for individual queries (client-side query settings), or for all queries in the workgroup. You cannot omit specifying the location altogether. If the location is not specified in either way, Athena issues an error at query execution. If you use the Athena console, and don't specify the query results location using one of the methods, Athena uses the default location (p. 84).

The client-side query settings are used only if your workgroup's settings, which include query results location and encryption, are not enforced. If your workgroup's settings override client-side settings, then the location and encryption configuration you specify in the Settings are not used. This applies to queries you run in the console, by using the API operations, or the driver. If Override client-side settings is selected, your query uses the workgroup's settings, even though the settings specified for this particular query may differ from the workgroup's settings. For more information, see Workgroup Settings OverrideClient-Side Settings (p. 166).

If you select Override client-side settings for the workgroup, the following screen displays when you choose Settings. This indicates that client-side settings are not used for queries in this workgroup.

Saving Query Results

After you run the query, the results appear in the Results pane.

To save the results of the most recent query to CSV, choose the file icon.

To save the results of a query you ran previously, choose History, locate your query, and use Download Results.

The Default Location for Query Results

Important
The default location for query results is used in the Athena console only if you run your queries in the console, have not set the results location for queries in the workgroup, and if the
workgroup settings do not override client-side settings. For more information, see Workgroup Settings Override Client-Side Settings (p. 166).

In this case only, Athena stores individual query results in this Amazon S3 bucket by default:

```
aws-athena-query-results-<ACCOUNTID>-<REGION>
```

**Note**
The default location is used only for queries you run in the Athena console. If you use the API or the drivers to run queries, you must specify the query results location using one of the ways: either for individual queries, using OutputLocation (client-side), or in the workgroup, using WorkGroupConfiguration.

Query results are saved based on the name of the query and the date the query ran, as follows:

```
{QueryLocation}/{QueryName|Unsaved}/{yyyy}/{mm}/{dd}/{QueryID}.csv
```

```
{QueryLocation}/{QueryName|Unsaved}/{yyyy}/{mm}/{dd}/{QueryID}.csv.metadata
```

In this notation:

- **QueryLocation** is the base location for all query results if the workgroup's settings are not used. To view or change this location, choose Settings in the upper right pane. You can enter a new value for Query result location at any time. You can also choose to encrypt individual query results in Amazon S3. For more information, see Configuring Encryption Options (p. 62).
- **QueryName** is the name of the query for which the results are saved. If the query wasn't saved, Unsaved appears. To see a list of queries and examine their SQL statements, choose Saved queries.
- **yyyy/mm/dd/** is the date the query ran.
- **QueryID** is the unique ID of the query.

## Viewing Query History

To view your recent query history, use History. Athena retains query history for 45 days.

**Note**
Starting on December 1, 2017, Athena retains query history for 45 days.

To retain query history for a longer period, write a program using methods from Athena API and the AWS CLI to periodically retrieve the query history and save it to a data store:

1. Retrieve the query IDs with ListQueryExecutions.
2. Retrieve information about each query based on its ID with GetQueryExecution.
3. Save the obtained information in a data store, such as Amazon S3, using the put-object AWS CLI command from the Amazon S3 API.

## Viewing Query History

1. To view a query in your history for up to 45 days after it ran, choose History and select a query. You can also see which queries succeeded and failed, download their results, and view query IDs, by clicking the status value.
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. 227).

Topics

- When to Use Views? (p. 86)
- Supported Actions for Views in Athena (p. 87)
- Considerations for Views (p. 87)
- Working with Views in the Console (p. 88)
- Creating Views (p. 89)
- Examples of Views (p. 90)
- Updating Views (p. 91)
- Deleting Views (p. 91)

When to Use Views?

You may want to create views to:

- Query a subset of data. For example, you can create a table 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. 227)</td>
<td>Creates a new view from a specified <code>SELECT</code> query. For more information, see Creating Views (p. 89). The optional <code>OR REPLACE</code> clause lets you update the existing view by replacing it.</td>
</tr>
<tr>
<td>DESCRIBE VIEW (p. 228)</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. 229)</td>
<td>Deletes an existing view. The optional <code>IF EXISTS</code> clause suppresses the error if the view does not exist. For more information, see Deleting Views (p. 91).</td>
</tr>
<tr>
<td>SHOW CREATE VIEW (p. 231)</td>
<td>Shows the SQL statement that creates the specified view.</td>
</tr>
<tr>
<td>SHOW VIEWS (p. 233)</td>
<td>Lists the views in the specified database, or in the current database if you omit the database name. Use the optional <code>LIKE</code> 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. 230)</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"],
        "Name":"b","Type":"string"
      ]
  }
```
Amazon Athena User Guide
Working with Views in the Console

- 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 detects stale views and displays an error message in such cases. A stale view is a view query that references tables or databases that do not exist.
- You can create and run nested views as long as the query behind the nested view is valid and the tables and databases exist.
- Athena view names cannot contain special characters, other than underscore (_). For more information, see Names for Tables, Databases, and Columns (p. 71).
- 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. 72).
- 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. 51).

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. 233) 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:

   ```
   "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": []
   ```

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:
3. Choose an option. For example, Show properties shows the view name, the name of the database in which the table for the view is created in Athena, and the time stamp when it was created:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>View name</td>
<td>employee_view</td>
</tr>
<tr>
<td>Database Name</td>
<td></td>
</tr>
<tr>
<td>Create Time</td>
<td>2018/03/07 19:08:33 UTC-5</td>
</tr>
</tbody>
</table>

## 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 button](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. 227) and Examples of Views (p. 90).

   View names cannot contain special characters, other than underscore (_). See Names for Tables, Databases, and Columns (p. 71). Avoid using Reserved Keywords (p. 72) 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. 231).

### Example Example 1

Consider the following two tables: a table `employees` with two columns, `id` and `name`, and a table `salaries`, with two columns, `id` and `salary`.

In this example, we create a view named `name_salary` as a `SELECT` query that obtains a list of IDs mapped to salaries from the tables `employees` and `salaries`:

```sql
CREATE VIEW name_salary AS
  SELECT employees.name,
         salaries.salary
  FROM employees, salaries
  WHERE employees.id = salaries.id
```

### Example Example 2

In the following example, we create a view named `view1` that enables you to hide more complex query syntax.

This view runs on top of two tables, `table1` and `table2`, where each table is a different `SELECT` query. The view selects all columns from `table1` and joins the results with `table2`. The join is based on column `a` that is present in both tables.

```sql
CREATE VIEW view1 AS
WITH
  table1 AS (SELECT a,
             MAX(b) AS b
             FROM x
             GROUP BY a
          ),
  table2 AS (SELECT a,
             AVG(d) AS d
             FROM y
             GROUP BY a
          )
SELECT table1.*, table2.*
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. 227).

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. 229).

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. 225).

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. 78).
- Create copies of existing tables that contain only the data you need.

Topics

- Considerations and Limitations for CTAS Queries (p. 91)
- Running CTAS Queries in the Console (p. 93)
- Bucketing vs Partitioning (p. 96)
- Examples of CTAS Queries (p. 97)

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. 225).</td>
</tr>
<tr>
<td>Note</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. 72), 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. 71).</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>
</tbody>
</table>
### Considerations and Limitations for CTAS Queries

<table>
<thead>
<tr>
<th>Item</th>
<th>What You Need to Know</th>
</tr>
</thead>
</table>
| **Location of CTAS query results** | The location for storing CTAS query results in Amazon S3 must be 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, otherwise your CTAS query will fail.  

You can specify the location for storing your CTAS query results. If omitted and if your workgroup does not override client-side settings (p. 166), Athena uses this location by default: `s3://aws-athena-query-results-<account>-<region>/<query-name-or-unsaved>/`.  

If your workgroup overrides client-side settings, this means that the workgroup’s query result location is used for your CTAS queries. If you specify a different results location, your query will fail. To obtain the results location specified for the workgroup, view workgroup’s details (p. 169).  

If the workgroup in which a query will run is configured with an enforced query results location (p. 166), 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; |
| **Formats for storing query results** | The results of CTAS queries are stored in Parquet by default, if you don’t specify a data storage format. You can store CTAS results in PARQUET, ORC, AVRO, JSON, and TEXTFILE. CTAS queries do not require specifying a SerDe to interpret format transformations. See Example 5: Storing Results of a CTAS Query in Another Format (p. 98). |
| **Compression formats** | GZIP compression is used for CTAS query results by default. For Parquet and ORC, you can also specify SNAPPY. See Example 4: Specifying Data Storage and Compression Formats for CTAS Query Results (p. 98). |
| **Partitioning** | You can partition the results data of a CTAS query by one or more columns. When creating a partitioned table, Athena automatically adds partitions to the AWS Glue Data Catalog.  

**Important**  
The results of a CTAS query in Athena can have a maximum of 100 partitions that Athena creates for you when writing CTAS query results to a specified location in Amazon S3. If the number of partitions to which the results data is written in parallel exceeds 100, Athena issues an error.  

List partition columns at the end of the SELECT statement in a CTAS query. For more information, see Example 7: CTAS Queries with Partitions (p. 99) and Bucketing vs Partitioning (p. 96). |
| **Bucketing** | You can configure buckets for storing the results of a CTAS query and bucket data by one or more columns. There is no limit to the number of buckets you can specify. For more information, see Example 8: A CTAS Query with Bucketing (p. 100) and Bucketing vs Partitioning (p. 96). |
| **Encryption** | You can encrypt CTAS query results in Amazon S3, similar to the way you encrypt other query results in Athena. For more information, see Configuring Encryption Options (p. 62). |
| **Data types** | Column data types for a CTAS query are the same as specified for the original query. |
Running CTAS Queries in the Console

In the Athena console, you can:

- Create a CTAS query from another query (p. 93)
- Create a CTAS query from scratch (p. 93)

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. 166), 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. 78).
Choose **Next** to review your query and revise it as needed. For query syntax, see CREATE TABLE AS (p. 225). The preview window opens, as shown in the following example:
g. Choose **Create**.

3. Choose **Run query**.

**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. 225).

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. 91).

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. In Athena, this number is up to 100 partitions. To choose the columns by which to partition your CTAS query results use the following tips:
  - 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 is 100.
  - 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.
  - 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 partition_by in CREATE TABLE AS (p. 225).

- **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.
  - You can specify any number of buckets for your CTAS query results, using one or more columns as bucket names. 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. For example, 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. 225).

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 ts (for storing timestamp type data), you can partition your CTAS query results by department and sales_quarter, because these columns have relatively low cardinality of values; for example, 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 `ts` timestamp type values stored in a `ts` column, you can configure bucketing for the same query results by the column `ts`. This column has high cardinality and the data from it can be distributed and stored in many buckets in Amazon S3. Consider an opposite scenario: if you don't create buckets for timestamp type data and run a query for particular date or time 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 date- and time-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.

**Examples of CTAS Queries**

Use the following examples to create CTAS queries. For information about the CTAS syntax, see [CREATE TABLE AS](p. 225).

In this section:
- **Example 1**: Selecting All Columns with CTAS (p. 97)
- **Example 2**: Selecting Specific Columns from One or More Tables with CTAS (p. 97)
- **Example 3**: Creating an Empty Copy of an Existing Table with CTAS (p. 98)
- **Example 4**: Specifying Data Storage and Compression Formats for CTAS Query Results (p. 98)
- **Example 5**: Storing Results of a CTAS Query in Another Format (p. 98)
- **Example 6**: CTAS Queries without Partitions (p. 98)
- **Example 7**: CTAS Queries with Partitions (p. 99)
- **Example 8**: A CTAS Query with Bucketing (p. 100)

**Example Example 1: Selecting All Columns with CTAS**

The following example creates a table by copying all columns from a table:

```sql
CREATE TABLE new_table AS
SELECT *
FROM old_table;
```

In the following variation of the same example, your `SELECT` statement also includes a `WHERE` clause. In this case, the query selects only those rows from the table that satisfy the `WHERE` clause:

```sql
CREATE TABLE new_table AS
SELECT *
FROM old_table
WHERE condition;
```

**Example Example 2: Selecting Specific Columns from One or More Tables with CTAS**

The following example creates a new query that runs on a set of columns from another table:

```sql
CREATE TABLE new_table AS
SELECT column_1, column_2, ... column_n
FROM old_table;
```

This variation of the same example creates a new table from specific columns from multiple tables:
CREATE TABLE new_table AS
SELECT column_1, column_2, ... column_n
FROM old_table_1, old_table_2, ... old_table_n;

Example Example 3: Creating an Empty Copy of an Existing Table with CTAS

The following example uses \texttt{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 4: Specifying Data Storage and Compression Formats for CTAS Query Results

The following example creates a new CTAS query that saves data in Parquet. This allows you to change the storage format from the one used by the original table. You can specify \texttt{PARQUET}, \texttt{ORC}, \texttt{AVRO}, \texttt{JSON}, and \texttt{TEXTFILE} in a similar way.

This example also specifies compression as \texttt{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 \texttt{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 5: Storing Results of a CTAS Query in Another Format

The following CTAS query takes the results from another query, which could be stored in CSV or another text format, and stores them in ORC:

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 6: CTAS Queries without Partitions

The following examples create tables that are not partitioned. The table data is stored in different formats. Some of these examples specify the external location.
The following example creates a CTAS query that stores the results as a text file:

```sql
CREATE TABLE ctas_csv_unpartitioned
WITH (format = 'TEXTFILE',
     external_location = 's3://my_athena_results/ctas_csv_unpartitioned/')
AS SELECT key1, name1, address1, comment1
FROM table1;
```

In the following example, results are stored in Parquet, and the default results location is used:

```sql
CREATE TABLE ctas_parquet_unpartitioned
WITH (format = 'PARQUET')
AS SELECT key1, name1, comment1
FROM table1;
```

In the following query, the table is stored in JSON, and specific columns are selected from the original table's results:

```sql
CREATE TABLE ctas_json_unpartitioned
WITH (format = 'JSON',
     external_location = 's3://my_athena_results/ctas_json_unpartitioned/')
AS SELECT key1, name1, address1, comment1
FROM table1;
```

In the following example, the format is ORC:

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

In the following example, the format is Avro:

```sql
CREATE TABLE ctas_avro_unpartitioned
WITH (format = 'AVRO',
     external_location = 's3://my_athena_results/ctas_avro_unpartitioned/')
AS SELECT key1, name1, comment1
FROM table1;
```

**Example Example 7: CTAS Queries with Partitions**

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. 226). For more information about choosing the columns for partitioning, see Bucketing vs Partitioning (p. 96).

**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 partitions.

```sql
CREATE TABLE ctas_csv_partitioned
WITH (format = 'TEXTFILE',
     external_location = 's3://my_athena_results/ctas_csv_partitioned/',
     partitioned_by = ARRAY['key1'])
```
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. 100)
- Concatenating Arrays (p. 102)
- Converting Array Data Types (p. 103)
- Finding Lengths (p. 103)
- Accessing Array Elements (p. 103)
- Flattening Nested Arrays (p. 104)
- Creating Arrays from Subqueries (p. 107)
- Filtering Arrays (p. 107)
- Sorting Arrays (p. 108)
- Using Aggregation Functions with Arrays (p. 108)
- Converting Arrays to Strings (p. 109)

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:

```
+-----------+
| items     |
+-----------+
| [1,2,3,4] |
+-----------+
```

This query creates two arrays.

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

It returns:

```
+--------------------+
| items              |
+--------------------+
| [[1, 2], [3, 4]]   |
+--------------------+
```

To create an array from selected columns of compatible types, use a query, as in this example:

```
WITH
dataset AS (  
  SELECT 1 AS x, 2 AS y, 3 AS z  
)
SELECT ARRAY [x,y,z] AS items FROM dataset
```

This query returns:

```
+-----------+
| items     |
+-----------+
| [1,2,3]   |
+-----------+
```

In the following example, two arrays are selected and returned as a welcome message.

```
WITH
  dataset AS (  
    SELECT  
      ARRAY ['hello', 'amazon', 'athena'] AS words,  
      ARRAY ['hi', 'alexa'] AS alexa  
  )
SELECT ARRAY[words, alexa] AS welcome_msg FROM dataset
```

This query returns:
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:

```sql
SELECT ARRAY[
    MAP(ARRAY['first', 'last', 'age'], ARRAY['Bob', 'Smith', '40']),
    MAP(ARRAY['first', 'last', 'age'], ARRAY['Jane', 'Doe', '30']),
    MAP(ARRAY['first', 'last', 'age'], ARRAY['Billy', 'Smith', '8'])
] AS people
```

This query returns:

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

Concatenating Arrays

To concatenate multiple arrays, use the double pipe `||` operator between them.

```sql
SELECT ARRAY [4,5] || ARRAY[ ARRAY[1,2], ARRAY[3,4] ] AS items
```

This query returns:

```
+--------------------------+
| items                    |
+--------------------------+
| [4, 5], [1, 2], [3, 4]   |
+--------------------------+
```

To combine multiple arrays into a single array, use the `concat` function.

```sql
WITH dataset AS (
    SELECT
        ARRAY ['hello', 'amazon', 'athena'] AS words,
        ARRAY ['hi', 'alexa'] AS alexa
    )
SELECT concat(words, alexa) AS welcome_msg
FROM dataset
```

This query returns:

```
+--------------------------+
| welcome_msg              |
+--------------------------+
| [hello, amazon, athena], [hi, alexa] |
+--------------------------+
```
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
```
This query returns:

```
<table>
<thead>
<tr>
<th>item_count</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
```

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:
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.

```
SELECT flatten(ARRAY[ ARRAY[1,2], ARRAY[3,4] ]) AS items
```
This query returns:

```
+-----------+
| items     |
+-----------+
| [1,2,3,4] |
+-----------+
```

To flatten an array into multiple rows, use `CROSS JOIN` in conjunction with the `UNNEST` operator, as in this example:

```sql
WITH dataset AS (  
  SELECT  
    'engineering' as department,  
    ARRAY['Sharon', 'John', 'Bob', 'Sally'] as users  
)  
SELECT department, names FROM dataset  
CROSS JOIN UNNEST(users) as t(names)
```

This query returns:

```
+----------------------+
| department | names  |
+----------------------+
| engineering | Sharon |
| engineering | John   |
| engineering | Bob    |
| engineering | Sally  |
+----------------------+
```

To flatten an array of key-value pairs, transpose selected keys into columns, as in this example:

```sql
WITH  
  dataset AS (  
    SELECT  
      'engineering' as department,  
      ARRAY[  
        MAP(ARRAY['first', 'last', 'age'],ARRAY['Bob', 'Smith', '40']),  
        MAP(ARRAY['first', 'last', 'age'],ARRAY['Jane', 'Doe', '30']),  
        MAP(ARRAY['first', 'last', 'age'],ARRAY['Billy', 'Smith', '8'])  
      ] AS people  
  )  
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.

```
WITH
dataset AS {
    SELECT ARRAY[
        CAST(ROW('Sally', 'engineering', ARRAY[1,2,3,4]) AS ROW(name VARCHAR, department VARCHAR, scores ARRAY(INTEGER))),
        CAST(ROW('John', 'finance', ARRAY[7,8,9]) AS ROW(name VARCHAR, department VARCHAR, scores ARRAY(INTEGER))),
        CAST(ROW('Amy', 'devops', ARRAY[12,13,14,15]) AS ROW(name VARCHAR, department VARCHAR, scores ARRAY(INTEGER)))
    ] AS users,

    users AS {
        SELECT person, score
        FROM
            dataset,
            UNNEST(dataset.users) AS t(person),
            UNNEST(person.scores) AS t(score)
    }

    SELECT person.name, person.department, SUM(score) AS total_score FROM users
    GROUP BY (person.name, person.department)
    ORDER BY (total_score) DESC
    LIMIT 1
}
```

This query returns:

```
<table>
<thead>
<tr>
<th>name</th>
<th>department</th>
<th>total_score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amy</td>
<td>devops</td>
<td>54</td>
</tr>
</tbody>
</table>
```

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

```
WITH
dataset AS {
    SELECT ARRAY[
        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, score FROM users
    ORDER BY (score) DESC
    LIMIT 1
}
```

This query returns:

```
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>106</td>
</tr>
</tbody>
</table>
```
Creating Arrays from Subqueries

Create an array from a collection of rows.

```sql
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:

<table>
<thead>
<tr>
<th>array_items</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1, 2, 3, 4, 5]</td>
</tr>
</tbody>
</table>

To create an array of unique values from a set of rows, use the `distinct` keyword.

```sql
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.

<table>
<thead>
<tr>
<th>array_items</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1, 2, 3, 4, 5]</td>
</tr>
</tbody>
</table>

Filtering Arrays

Create an array from a collection of rows if they match the filter criteria.

```sql
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:
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>

Sorting Arrays

Create a sorted array of unique values from a set of rows.

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:

<table>
<thead>
<tr>
<th>array_items</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1, 2, 3, 4, 5, 6]</td>
</tr>
</tbody>
</table>

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. 107).
Note
ORDER BY is not supported for aggregation functions, for example, you cannot use it within array_agg(x).

WITH
dataset AS {
    SELECT ARRAY
        [
            ARRAY[1,2,3,4],
            ARRAY[5,6,7,8],
            ARRAY[9,0]
        ] AS items
},
item AS {
    SELECT i AS array_items
    FROM dataset, UNNEST(items) AS t(i)
}
SELECT array_items, sum(val) AS total
FROM item, UNNEST(array_items) AS t(val)
GROUP BY array_items

This 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.

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>
<tbody>
<tr>
<td>hello amazon athena</td>
</tr>
</tbody>
</table>
+---------------------+

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

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:

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>users</td>
</tr>
<tr>
<td>-------------------------</td>
</tr>
<tr>
<td>{field0=Bob, field1=38}</td>
</tr>
<tr>
<td>-------------------------</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>users</td>
</tr>
<tr>
<td>-------------------------</td>
</tr>
<tr>
<td>{NAME=Bob, AGE=38}</td>
</tr>
<tr>
<td>-------------------------</td>
</tr>
</tbody>
</table>

Note
In the example above, you declare name as a VARCHAR because this is its type in Presto. If you declare this STRUCT inside a CREATE_TABLE statement, use String type because Hive defines this data type as String.
Filtering Arrays Using the . Notation

In the following example, select the accountId field from the userIdentity column of a AWS CloudTrail logs table by using the dot . notation. For more information, see Querying AWS CloudTrail Logs (p. 135).

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

This query returns:

```
+--------------+
| newid        |
+--------------+
| 112233445566 |
| 99887665544  |
+--------------+
```

To query an array of values, issue this query:

```
WITH dataset AS (
  SELECT ARRAY[
    CAST(ROW('Bob', 38) AS ROW(name VARCHAR, age INTEGER)),
    CAST(ROW('Alice', 35) AS ROW(name VARCHAR, age INTEGER)),
    CAST(ROW('Jane', 27) AS ROW(name VARCHAR, age INTEGER))
  ] AS users
)
SELECT * FROM dataset
```

It returns this result:

```
+-----------------------------------------------------------------+
<table>
<thead>
<tr>
<th>users</th>
</tr>
</thead>
<tbody>
<tr>
<td>[{NAME=Bob, AGE=38}, {NAME=Alice, AGE=35}, {NAME=Jane, AGE=27}]</td>
</tr>
</tbody>
</table>
+-----------------------------------------------------------------+
```

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:

```
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. 104).

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:

```sql
WITH dataset AS (  
SELECT ARRAY[
  CAST(
    ROW('aws.amazon.com', ROW[
      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
```
Finding Keywords in Arrays Using `regexp_like`

```sql
FROM sites, UNNEST(sites.flaggedActivity.flags) t(flags)
WHERE regexp_like(flags['term'], 'politics|bigdata')
GROUP BY (hostname)
```

This query returns two sites:

<table>
<thead>
<tr>
<th>hostname</th>
</tr>
</thead>
<tbody>
<tr>
<td>aws.amazon.com</td>
</tr>
<tr>
<td>news.cnn.com</td>
</tr>
</tbody>
</table>

**Example Example 2: Using `regexp_like`**

The query in the following example adds up the total popularity scores for the sites matching your search terms with the `regexp_like` function, and then orders them from highest to lowest.

```sql
WITH dataset AS (  
SELECT 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:
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:

```
<table>
<thead>
<tr>
<th>user</th>
</tr>
</thead>
<tbody>
<tr>
<td>{last=Smith, first=Bob, age=35}</td>
</tr>
</tbody>
</table>
```

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:

```
<table>
<thead>
<tr>
<th>first_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
</tr>
</tbody>
</table>
```
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. 116)
- Extracting Data from JSON (p. 117)
- Searching for Values (p. 119)
- Obtaining Length and Size of JSON Arrays (p. 121)

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. 202).

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. 202).
- 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. 202).

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. 117)
- Converting JSON to Athena Data Types (p. 117)
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
```

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'

```sql
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.

```json
{"name": "Susan Smith",
"org": "engineering",
"projects":
```
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.

```sql
WITH dataset AS (  
  SELECT '{"name": "Susan Smith",  
          "org": "engineering",  
          "projects": [{"name":"project1", "completed":false},  
                         {"name":"project2", "completed":true}]}'  
     AS blob  
)
SELECT  
  json_extract(blob, '$.name') AS name,  
  json_extract(blob, '$.projects') AS projects  
FROM dataset
```

The returned value is a JSON-encoded string, and not a native Athena data type.

```
+----------------------------------------------------------+----------------------------------------------------------+
|     name         |     projects                                          |
|----------------------------------------------------------+----------------------------------------------------------|
| "Susan Smith"  | [{"name":"project1","completed":false},  |
|               | {"name":"project2","completed":true}] |
|----------------------------------------------------------+----------------------------------------------------------|
```

To extract the scalar value from the JSON string, use the `json_extract_scalar` function. It is similar to `json_extract`, but returns only scalar values (Boolean, number, or string).

```sql
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:
To obtain the first element of the `projects` property in the example array, use the `json_array_get` function and specify the index position.

```
WITH dataset AS (
  SELECT '{"name": "Bob Smith",
         "org": "engineering",
         "projects": [{"name":"project1", "completed":false},{"name":"project2",
         "completed":true}]}'
  AS blob
SELECT json_array_get(json_extract(blob, '$.projects'), 0) AS item
FROM dataset
```

It returns the value at the specified index position in the JSON-encoded array.

```
<table>
<thead>
<tr>
<th>item</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;project1&quot;</td>
</tr>
</tbody>
</table>
```

To return an Athena string type, use the `[ ]` operator inside a JSONPath expression, then use the `json_extract_scalar` function. For more information about `[ ]`, see Accessing Array Elements (p. 103).

```
WITH dataset AS (  
  SELECT '{"name": "Bob Smith",
           "org": "engineering",
           "projects": [{"name":"project1", "completed":false},{"name":"project2",
           "completed":true}]}'
  AS blob
)  
SELECT json_extract_scalar(blob, '$.projects[0].name') AS project_name
FROM dataset
```

It returns this result:

```
<table>
<thead>
<tr>
<th>project_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>project1</td>
</tr>
</tbody>
</table>
```

### Searching for Values

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 '{"name": "Bob Smith",
  
```

```
SELECT * FROM (VALUES 
    (JSON '{"name": "Bob Smith", "org": "legal", "projects": ["project1"]}'),
    (JSON '{"name": "Susan Smith", "org": "engineering", "projects": ["project1", "project2", "project3"]}'),
    (JSON '{"name": "Jane Smith", "org": "finance", "projects": ["project1", "project2"]}')
) AS t (users)
)
SELECT json_extract_scalar(users, '$.name') AS user
FROM dataset
WHERE json_array_contains(json_extract(users, '$.projects'), 'project2')

This query returns a list of users.

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>user</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>Susan Smith</td>
</tr>
<tr>
<td>Jane Smith</td>
</tr>
<tr>
<td>-------------</td>
</tr>
</tbody>
</table>

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": ["project1", "completed":false]}'),
        (JSON '{"name": "Susan Smith", "org": "engineering", "projects": ["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, 'name') AS ARRAY (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
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.

```sql
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.

```sql
WITH dataset AS (  
SELECT * FROM (VALUES  
(JSON '{"name": "Bob Smith",  
"org": "legal",  
"projects": [{"name": "project1", "completed":false}]})'),  
(JSON '{"name": "Susan Smith",  
"org": "engineering",  
"projects": [{"name": "project2", "completed":true},  
{"name": "project3", "completed":true}]})'),  
(JSON '{"name": "Jane Smith",  
"org": "finance",  
"projects": [{"name": "project2", "completed":true}]})')  
) AS t (users)  
SELECT  
json_extract_scalar(users, '$.name') as name,  
json_size(json_extract(users, '$.projects')) as count  
FROM dataset  
ORDER BY count DESC
```

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>
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>
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. 123)
- Input Data Formats and Geometry Data Types (p. 123)
- List of Supported Geospatial Functions (p. 124)
- Examples: Geospatial Queries (p. 132)

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 a pair of double values for the geographic coordinates of Mount Rainier in Athena, use the `ST_POINT (double, double) (longitude, latitude)` geospatial function, specifying the longitude first, then latitude:

```
ST_POINT(-121.7602, 46.8527) (longitude, latitude)
```

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. 202).

**Geometry Data Types**

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

- point
- line
- polygon
- multiline
- multipolygon

**List of 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. This topic lists only the ESRI geospatial functions that are supported in Athena.
- You cannot use views with geospatial functions.

Athena supports four types of geospatial functions:

- **Constructor Functions** (p. 125)
- **Geospatial Relationship Functions** (p. 127)
- **Operation Functions** (p. 128)
- **Accessor Functions** (p. 130)

**Before You Begin**

Create two tables, `earthquakes` and `counties`, as follows:

```sql
CREATE external TABLE earthquakes
(
    earthquake_date STRING,
    latitude DOUBLE,
    longitude DOUBLE,
    depth DOUBLE,
    magnitude DOUBLE,
    magtype string,
    mbstations string,
    gap string,
    distance string,
    rms string,
    source string,
```
eventid string
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
STORED AS TEXTFILE LOCATION 's3://my-query-log/csv'

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'

Some of the subsequent examples are based on these tables and rely on two sample files stored in the Amazon S3 location. These files are not included with Athena and are used for illustration purposes only:

- An earthquakes.csv file, which lists earthquakes that occurred in California. This file has fields that correspond to the fields in the table earthquakes.
- A california-counties.json file, which lists JSON-encoded county data in the ESRI-compliant format, and includes many fields, such as AREA, PERIMETER, STATE, COUNTY, and NAME. The counties table is based on this file and has two fields only: Name (string), and BoundaryShape (binary).

**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_POINT(double, double)**

Returns 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 0.072284, which is the buffer distance, is specified in angular units as decimal degrees:

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

**Syntax:**

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

In the alternative syntax, you can also specify the coordinates as a point data type with two values:

```
SELECT ST_POINT('point (-74.006801 40.705220)');
```

**Example.** This example uses specific longitude and latitude coordinates from earthquakes.csv:
SELECT ST_POINT(61.56, -158.54) 
FROM earthquakes 
LIMIT 1;

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

00 00 00 00 01 01 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:

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:

SELECT ST_GEOMETRY_TO_TEXT(ST_POINT(-74.006801, 40.705220)) AS WKT;

This query returns a WKT representation of the point geometry type: 1 POINT (-74.006801 40.70522).

**ST_LINE(varchar)**

Returns a value in the line data type, which is a binary representation of the geometry data type line. Example:

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

**ST_POLYGON(varchar)**

Returns a value in the polygon data type, which is a binary representation of the geometry data type polygon. Example:

SELECT ST_Polygon('polygon ((1 1, 4 1, 1 4))')

**ST_GEOMETRY_TO_TEXT (varbinary)**

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

SELECT ST_GEOMETRY_TO_TEXT(ST_POINT(61.56, -158.54))

**ST_GEOMETRY_FROM_TEXT (varchar)**

Converts text into a geometry data type. Returns a value in a geometry 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)))
Geospatial Relationship Functions

The following functions express relationships between two different geometries that you specify as input. They 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 0 -1,0 2))', 'POLYGON((-1 3,2 1,0 -3,-1 3))')`

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

- `SELECT ST_CONTAINS(ST_GEOMETRY_FROM_TEXT('POLYGON((0 2,1 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:

- `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:

- `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:

- `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:

- `SELECT ST_INTERSECTS(ST_LINE('linestring(8 7, 7 8)'), ST_POLYGON('polygon((1 1, 4 1, 4 4, 1 4))'))`
ST_OVERLAPS (geometry, geometry)

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

```
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:

```
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:

```
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:

```
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.

All operation functions take as an input one of the geometry data types and return their binary representations.

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:

```
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, 1 4, 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, 1 5, 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))')))
```
**ST_SYMMETRIC_DIFFERENCE (geometry, geometry)**

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

```
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, multilineline, 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:

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

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

```
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 types, and returns the count of coordinate components in type bigint. Example:

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

**ST_DIMENSION (geometry)**

Takes as an input one of the supported geometry types, and returns the spatial dimension of a geometry in type bigint. Example:

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

**ST_DISTANCE (geometry, geometry)**

Returns the distance in type double between the left geometry and the right geometry. Example:
SELECT ST_DISTANCE(ST_POINT(0.0,0.0), ST_POINT(3.0,4.0))

**ST_IS_CLOSED (geometry)**

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

SELECT ST_IS_CLOSED(ST_LINE('linestring(0 2 2)'))

**ST_IS_EMPTY (geometry)**

Returns TRUE (type boolean) if and only if the specified geometry is empty. Example:

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:

SELECT ST_IS_RING(ST_LINE('linestring(0 2 2)'))

**ST_LENGTH (geometry)**

Returns the length of line in type double. Example:

SELECT ST_LENGTH(ST_LINE('linestring(0 2 2)'))

**ST_MAX_X (geometry)**

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

SELECT ST_MAX_X(ST_LINE('linestring(0 2 2)'))

**ST_MAX_Y (geometry)**

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

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

**ST_MIN_X (geometry)**

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

SELECT ST_MIN_X(ST_LINE('linestring(0 2 2)'))

**ST_MIN_Y (geometry)**

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

SELECT ST_MIN_Y(ST_LINE('linestring(0 2 2)'))
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_END_POINT (geometry)
Returns the last point of a line geometry data type in type point. Example:

```
SELECT ST_END_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))
```

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

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

Examples: Geospatial Queries

The following examples create two tables and issue a query against them.

Note
These files are not included with the product and are used in the documentation for illustration purposes only. They contain sample data and are not guaranteed to be accurate.

These examples rely on two files:
- An earthquakes.csv sample file, which lists earthquakes that occurred in California. This file has fields that correspond to the fields in the table earthquakes in the following example.
- A california-counties.json file, which lists JSON-encoded county data in the ESRI-compliant format, and includes many fields such as AREA, PERIMETER, STATE, COUNTY, and NAME. The
The following example shows the `counties` table from this file with two fields only: `Name` (string), and `BoundaryShape` (binary).

For additional examples of geospatial queries, see these blog posts:

- Querying OpenStreetMap with Amazon Athena
- Visualize over 200 years of global climate data using Amazon Athena and Amazon QuickSight.

The following code example creates a table called `earthquakes`:

```sql
CREATE external TABLE earthquakes
(
  earthquake_date string,
  latitude double,
  longitude double,
  depth double,
  magnitude double,
  magtype string,
  mbstations string,
  gap string,
  distance string,
  rms string,
  source string,
  eventid string
) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE LOCATION 's3://my-query-log/csv/';
```

The following code example creates a table called `counties`:

```sql
CREATE external TABLE IF NOT EXISTS counties
(
  Name string,
  BoundaryShape binary
```

The following code example uses the `CROSS JOIN` function for the two tables created earlier. Additionally, for both tables, it uses `ST_CONTAINS` and asks for counties whose boundaries include a geographical location of the earthquakes, specified with `ST_POINT`. It then groups such counties by name, orders them by count, and returns them in descending order.

```sql
SELECT counties.name,
       COUNT(*) cnt
FROM counties
CROSS JOIN earthquakes
WHERE ST_CONTAINS (counties.boundaryshape, ST_POINT(earthquakes.longitude,
  earthquakes.latitude))
GROUP BY counties.name
ORDER BY cnt DESC
```

This query returns:

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>------------------------</td>
</tr>
<tr>
<td>County</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Kern</td>
</tr>
<tr>
<td>San Bernardino</td>
</tr>
<tr>
<td>Imperial</td>
</tr>
<tr>
<td>Inyo</td>
</tr>
<tr>
<td>Los Angeles</td>
</tr>
<tr>
<td>Riverside</td>
</tr>
<tr>
<td>Monterey</td>
</tr>
<tr>
<td>Santa Clara</td>
</tr>
<tr>
<td>San Benito</td>
</tr>
<tr>
<td>Fresno</td>
</tr>
<tr>
<td>San Diego</td>
</tr>
<tr>
<td>Santa Cruz</td>
</tr>
<tr>
<td>Ventura</td>
</tr>
<tr>
<td>San Luis Obispo</td>
</tr>
<tr>
<td>Orange</td>
</tr>
<tr>
<td>San Mateo</td>
</tr>
</tbody>
</table>
Querying AWS Service Logs

This section includes several procedures for using Amazon Athena to query popular datasets, such as AWS CloudTrail logs, Amazon CloudFront logs, Classic Load Balancer logs, Application Load Balancer logs, Amazon VPC flow logs, and Network Load Balancer logs.

The tasks in this section use the Athena console, but you can also use other tools that connect via JDBC. For more information, see Using Athena with the JDBC Driver (p. 43), the AWS CLI, or the Amazon Athena API Reference.

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. 21) and Getting Started (p. 23).

Topics
- Querying AWS CloudTrail Logs (p. 135)
- Querying Amazon CloudFront Logs (p. 139)
- Querying Classic Load Balancer Logs (p. 140)
- Querying Network Load Balancer Logs (p. 142)
- Querying Application Load Balancer Logs (p. 143)
- Querying Amazon VPC Flow Logs (p. 145)

Querying AWS CloudTrail Logs

AWS CloudTrail is a service that records AWS API calls and events for AWS 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. 136)
- Creating a Table for CloudTrail Logs in the CloudTrail Console (p. 136)
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.gzipped). 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 for the 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.

Creating a Table for CloudTrail Logs in the CloudTrail Console

You can automatically create tables for querying CloudTrail logs directly from the CloudTrail console. This is a fairly straightforward method of creating tables, but you can only create tables this way if the Amazon S3 bucket that contains the log files for the trail is in a Region supported by Amazon Athena, and you are logged in with an IAM user or role that has sufficient permissions to create tables in Athena. For more information, see Setting Up (p. 21).

To create a table for a CloudTrail trail in the CloudTrail console

1. Open the CloudTrail console at https://console.aws.amazon.com/cloudtrail/.
2. In the navigation pane, choose **Event history**.
3. In **Event history**, choose **Run advanced queries in Amazon Athena**.
4. For **Storage location**, choose the Amazon S3 bucket where log files are stored for the trail to query.
   
   **Note**
   
   You can find out what bucket is associated with a trail by going to **Trails** and choosing the trail. The bucket name is displayed in **Storage location**.
5. Choose **Create table**. The table is created with a default name that includes the name of the Amazon S3 bucket.

## Manually Creating the Table for CloudTrail Logs in Athena

You can manually create tables for CloudTrail log files in the Athena console, and then run queries in Athena.

### To create a table for a CloudTrail trail in the CloudTrail console

1. Copy and paste the following DDL statement into the Athena console.
2. Modify the `s3://CloudTrail_bucket_name/AWSLogs/Account_ID/` to point to the Amazon S3 bucket that contains your logs 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. 117).

```sql
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,
)  
```
Example Query for CloudTrail Logs

The following example shows a portion of a query that returns all anonymous (unsigned) requests from the table created on top of CloudTrail event logs. This query selects those requests where useridentity.accountid is anonymous, and useridentity.arn is not specified:

```
SELECT *
FROM cloudtrail_logs
WHERE
  eventsource = 's3.amazonaws.com' AND
  eventname in ('GetObject') AND
  useridentity.accountid 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 "Manually Creating the Table for CloudTrail Logs in Athena" (p. 137). 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. 137).

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. 117).

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.

```sql
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 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. 139)
- Example Query for CloudFront logs (p. 140)

### Creating the Table for CloudFront Logs

**To create the CloudFront 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.

   This query uses the LazySimpleSerDe (p. 205) by default and it is omitted.

   The column `date` is escaped using backticks (´) because it is a reserved word in Athena. For information, see Reserved Keywords (p. 72).

```sql
CREATE EXTERNAL TABLE IF NOT EXISTS default.cloudfront_logs (  
    `date` DATE,
    time STRING,
    location STRING,
    bytes BIGINT,
    request_ip STRING,
    method STRING,
    host STRING,
    uri STRING,
    status INT,
    referrer STRING,
);```
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;
```

In some cases, you need to eliminate empty values from the results of `CREATE TABLE` query for CloudFront. To do so, run:

```
SELECT DISTINCT *
FROM cloudfront_logs
LIMIT 10;
```

For more information, see the AWS Big Data Blog post Build a Serverless Architecture to Analyze Amazon CloudFront Access Logs Using AWS Lambda, Amazon Athena, and Amazon Kinesis Analytics.

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. 141)
- Elastic Load Balancing Example Queries (p. 141)
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]*)([^ ]*)([\d\-]*)([^ ]*) ([\d\-]*)([^ ]*) ([\d\-]*)([^ ]*) ([\d\-]*)([^ ]*) ([\d\-]*)([^ ]*) ([\d\-]*)([^ ]*) ([\d\-]*)([^ ]*) ([\d\-]*)([^ ]*) ([\d\-]*)([^ ]*) ([\d\-]*)([^ ]*) ([\d\-]*)([^ ]*) ([\d\-]*)([^ ]*) ([\d\-]*)([^ ]*) ([\d\-]*)([^ ]*) ([\d\-]*)([^ ]*) ([\d\-]*)([^ ]*) ([\d\-]*)([^ ]*)'  
)  
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. 141)

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.

```sql
SELECT  
  request_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.
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. 142)
- Network Load Balancer Example Queries (p. 143)

To create the table for Network Load Balancer logs

1. Copy and paste the following DDL statement into the Athena console. Check the syntax of the Network Load Balancer log records. You may need to update the following query to include the columns and the Regex syntax for latest version of the record.

```sql
CREATE EXTERNAL TABLE IF NOT EXISTS nlb_tls_logs
(type string,
 version string,
 time string,
 elb string,
 listener_id string,
 client_ip string,
 client_port int,
 target_ip string,
 target_port int,
 tcp_connection_time_ms double,
 tls_handshake_time_ms double,
 received_bytes bigint,
 sent_bytes bigint,
 incoming_tls_alert int,
 cert_arn string,
 certificate_serial string,
 tls_cipher_suite string,
 tls_protocol_version string,
 tls_named_group string,
 domain_name string,
 new_field string)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
'serialization.format' = '1',
'input.regex' =
'([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*)')
```
LOCATION 's3://your_log_bucket/prefix/AWSLogs/AWS_account_ID/elasticloadbalancing/region';

2. Modify the LOCATION Amazon S3 bucket to specify the destination of your Network Load Balancer logs.

3. Run the query in the Athena console. After the query completes, Athena registers the nlb_tls_logs table, making the data in it ready for queries.

Network Load Balancer Example Queries

To see how many times a certificate is used, use a query similar to this example:

```
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:

```
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:

```
SELECT *
FROM "nlb_tls_logs"
ORDER BY tls_handshake_time_ms DESC
LIMIT 10;
```

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.

Before you begin, enable access logging for Application Load Balancer logs to be saved to your Amazon S3 bucket.

- Creating the Table for ALB Logs (p. 143)
- Example Queries for ALB logs (p. 144)

Creating the Table for ALB Logs

1. Copy and paste the following DDL statement into the Athena console, and modify values in LOCATION 's3://your-alb-logs-directory/AWSLogs/<ACCOUNT-ID>/elasticloadbalancing/<REGION>'.
Create the `alb_logs` table as follows.

**Note**
This query includes all fields present in the list of current Application Load Balancer Access Log Entries. It also includes a table column `new_field` at the end, in case you require additions to the ALB logs. This field does not break your query. The regular expression in the SerDe properties ignores this field if your logs don't have it.

```sql
CREATE EXTERNAL TABLE IF NOT EXISTS alb_logs (
  type string,
  time string,
  elb string,
  client_ip string,
  client_port int,
  target_ip string,
  target_port int,
  request_processing_time double,
  target_processing_time double,
  response_processing_time double,
  elb_status_code 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,
  domain_name string,
  chosen_cert_arn string,
  matched_rule_priority string,
  request_creation_time string,
  actions_executed string,
  redirect_url string,
  lambda_error_reason string,
  new_field string
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
  'serialization.format' = '1',
  'input.regex' =
  '([^\s]*) \([^\s]*\) \([^\s]*\) \([^\s]*\):([0-9]*) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \([^\s]*\) \[^\s]*\)
LOCATION 's3://your-alb-logs-directory/AWSLogs/<ACCOUNT-ID>/elasticsearchloadbalancing/<REGION>';"
)

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,
```
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.

- Creating the Table for VPC Flow Logs (p. 145)
- Example Queries for Amazon VPC Flow Logs (p. 146)

## Creating the Table for VPC Flow Logs

To create the Amazon VPC table:

1. Copy and paste the following DDL statement into the Athena console. This 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. 205).` In addition, in this query, fields are terminated by a space.

2. Modify the `LOCATION 's3://your_log_bucket/prefix/AWSLogs/ {subscribe_account_id}/vpcflowlogs/{region_code}/'` to point to the Amazon S3 bucket that contains your log data.

```sql
CREATE EXTERNAL TABLE IF NOT EXISTS vpc_flow_logs (dt string, version int, account string, interfaceid string, sourceaddress string, destinationaddress string,
```

---

### Querying Amazon VPC Flow Logs

```sql
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_config_us
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;
```
Example Queries for Amazon VPC Flow Logs

The following query lists all of the rejected TCP connections and uses the newly created date partition column, dt, to extract from it the day of the week for which these events occurred.

```sql
SELECT day_of_week(from_iso8601_timestamp(dt)) AS day,
       dt,
       interfaceid,
       sourceaddress,
       action,
       protocol
FROM vpc_flow_logs
WHERE action = 'REJECT' AND protocol = 6
LIMIT 100;
```

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.

```sql
SELECT SUM(numpackets) AS
```
packetcount, destinationaddress
FROM vpc_flow_logs
WHERE destinationport = 443
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.
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.

Important
Schema updates described in this section do not work on tables with complex or nested data types, such as arrays and structs.

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. 148)
- Index Access in ORC and Parquet (p. 149)
- Types of Updates (p. 151)
- Updates in Tables with Partitions (p. 156)

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. 149).

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 (default)</th>
<th>ORC: Read by Index</th>
<th>ORC: Read by Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename columns (p. 153)</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>Y</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. 152)</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. 152)</td>
<td>Store your data in CSV or TSV, JSON, AVRO, and in ORC and Parquet if they are read by name.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Remove columns (p. 153)</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>N</td>
</tr>
<tr>
<td>Reorder columns (p. 154)</td>
<td>Store your data in AVRO, JSON or ORC and Parquet if they are read by name.</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Change a column's data type (p. 155)</td>
<td>Store your data in any format, but test your query in Athena to make sure the data types are compatible. For Parquet and ORC, changing a data type works only for partitioned tables.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</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. 150).

• 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. 150).

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.

The following sections describe these cases in detail.

**ORC: Read by Index**

A table in **ORC is read by index**, by default. This is defined by the following syntax:

```sql
WITH SERDEPROPERTIES ( 'orc.column.index.access'='true' )
```

**Reading by index** allows you to rename columns. But then you lose the ability to remove columns or add
them in the middle of the table.

To make ORC read by name, which will allow you to add columns in the middle of the table or remove
columns in ORC, set the SerDe property **orc.column.index.access** to **FALSE** in the **CREATE TABLE**
statement. In this configuration, you will lose the ability to rename columns.

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.

**Important**

Schema updates described in this section do not work on tables with complex or nested data types, such as arrays and structs.

- Adding Columns at the Beginning or Middle of the Table (p. 152)
- Adding Columns at the End of the Table (p. 152)
- Removing Columns (p. 153)
- Renaming Columns (p. 153)
- Reordering Columns (p. 154)
- Changing a Column’s Data Type (p. 155)

Depending on how you expect your schemas to evolve, to continue using Athena queries, choose a compatible data format.

Let’s consider an application that reads orders information from an `orders` table that exists in two formats: CSV and Parquet.

The following example creates a table in Parquet:

```sql
CREATE EXTERNAL TABLE orders_parquet (  
  `orderkey` int,  
  `orderstatus` string,  
  `totalprice` double,  
  `orderdate` string,  
  `orderpriority` string,  
  `clerk` string,  
  `shippriority` int
) STORED AS PARQUET  
LOCATION 's3://schema_updates/orders_parquet/';
```

The following example creates the same table in CSV:

```sql
CREATE EXTERNAL TABLE orders_csv (  
  `orderkey` int,  
  `orderstatus` string,  
  `totalprice` double,  
  `orderdate` string,  
  `orderpriority` string,  
  `clerk` string,  
  `shippriority` int
)  
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',',  
LOCATION 's3://schema_updates/orders_csv/';
```

In the following sections, we review how updates to these tables affect Athena queries.
Adding Columns at the Beginning or in the Middle of the Table

Adding columns is one of the most frequent schema changes. For example, you may add a new column to enrich the table with new data. Or, you may add a new column if the source for an existing column has changed, and keep the previous version of this column, to adjust applications that depend on them.

To add columns at the beginning or in the middle of the table, and continue running queries against existing tables, use AVRO, JSON, and Parquet and ORC if their SerDe property is set to read by name. For information, see Index Access in ORC and Parquet (p. 149).

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,
    `o_orderkey` string
)
ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'
LOCATION 's3://schema_updates/orders_json/';
```

Adding Columns at the End of the Table

If you create tables in any of the formats that Athena supports, such as Parquet, ORC, Avro, JSON, CSV, and TSV, you can add new columns at the end of the table. If you use ORC formats, you must configure ORC to read by name. Parquet reads by name by default. For information, see Index Access in ORC and Parquet (p. 149).

In the following example, drop an existing table in Parquet, and add a new Parquet table with a new `comment` column at the end of the table:

```
DROP TABLE orders_parquet;
CREATE EXTERNAL TABLE orders_parquet (
    `orderkey` int,
    `orderstatus` string,
    `totalprice` double,
    `orderdate` string,
    `orderpriority` string,
    `clerk` string,
    `shippriority` int,
    `comment` string
)
STORED AS PARQUET
LOCATION 's3://schema_updates/orders_parquet/';
```

In the following example, drop an existing table in CSV and add a new CSV table with a new `comment` column at the end of the table:
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. 149).
- 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:

```sql
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. 149).

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.

In this example, rename the column `o_totalprice` to `o_total_price` in the Parquet table, and then run a query in Athena:
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://TBD/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:

```
SELECT *
FROM orders_parquet_column_renamed;
```

A query with a table in CSV looks similar:

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:

```
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 ORC, which reads by name by default. You can also make Parquet read by name, if needed. For information, see Index Access in ORC and Parquet (p. 149).

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,
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. 156).

Important
We strongly suggest that you test and verify your queries before performing data type translations. If Athena cannot convert the data type from the original data type to the target data type, the CREATE TABLE query may fail.

The following table lists data types that you can change:

<table>
<thead>
<tr>
<th>Original Data Type</th>
<th>Available Target Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING</td>
<td>BYTE, TINYINT, SMALLINT, INT, BIGINT</td>
</tr>
<tr>
<td>BYTE</td>
<td>TINYINT, SMALLINT, INT, BIGINT</td>
</tr>
<tr>
<td>TINYINT</td>
<td>SMALLINT, INT, BIGINT</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>INT, BIGINT</td>
</tr>
<tr>
<td>INT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>FLOAT</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

In the following example of the orders_json table, change the data type for the column `o_shippriority` to BIGINT:

```
CREATE EXTERNAL TABLE orders_json (  
  `o_orderkey` int,
  `o_custkey` int,  
  `o_orderstatus` string,
  `o_totalprice` double,
  `o_orderdate` string,  
  `o_orderpriority` string,
  `o_clerk` string,  
  `o_shippriority` BIGINT
)
ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'
```
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. 148).

Important

Schema updates described in this section do not work on tables with complex or nested data types, such as arrays and structs.

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.
Using Workgroups to Control Query Access and Costs

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 resources, you can use resource-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.

Topics
- Using Workgroups for Running Queries (p. 158)
- Controlling Costs and Monitoring Queries with CloudWatch Metrics (p. 174)

Using Workgroups for Running Queries

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. 159). 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. 165).</th>
</tr>
</thead>
</table>
| Enforce costs constraints. | You can set two types of cost constraints for queries in a workgroup:  
  - **Per-query limit** is a threshold for the amount of data scanned for each query. Athena cancels queries when they exceed the specified threshold. The limit applies to each running query within a workgroup. You can set only one per-query limit and update it if needed.  
  - **Per-workgroup limit** is a threshold you can set for each workgroup for the amount of data scanned by queries in the workgroup. Breaching a threshold activates an Amazon SNS alarm that triggers an action of your choice, such as sending an email to a specified user. You can set multiple per-workgroup limits for each workgroup.  
  For detailed steps, see Setting Data Usage Control Limits (p. 176). |
| Track query-related metrics for all workgroup queries in CloudWatch. | For each query that runs in a workgroup, if you configure the workgroup to publish metrics, Athena publishes Query Type, Query Execution Time, Query Status, Data Processed Per Query, and Workgroup Name to CloudWatch. You can also view query metrics (p. 174) for each of your workgroups within the Athena console.  
  In CloudWatch, you can create custom dashboards, and set thresholds and alarms on these metrics. |
How Workgroups Work

Workgroups in Athena have the following characteristics:

- By default, each account has a primary workgroup and the default permissions allow all authenticated users access to this workgroup. The primary workgroup cannot be deleted.

- You can create new workgroups. 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.

- 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.

- 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 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.

- Workgroups are IAM resources. You can apply resource-based policies to restrict access to the workgroup.

- 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.

- 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, if its users have access only to this workgroup, they lose access to running queries. Therefore, before deleting a workgroup, make sure its users have access to other workgroups in which they can continue to run queries. The primary workgroup cannot be deleted.

- 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. 166).

Workgroups integrate with IAM, CloudWatch, and Amazon Simple Notification Service. IAM 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.

**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. 170) to a workgroup you intend to use.

By default, if you have not created any workgroups, all queries in your account run in the primary workgroup:

> Athena Query Editor Saved Queries History AWS Glue Data Catalog Workgroup: primary

**High-Level Procedure for Setting up Workgroups**

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
</table>
| Step 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.  
  - 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.  
  - 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. |
| Step 2: Create workgroups as needed, and add tags to them. | Open the Athena console, choose the **Workgroup:<workgroup_name>** Workgroup (p. 167). |
| Step 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 example JSON policies, see Workgroup Example Policies (p. 59). |
| Step 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. 165). **Important**  
If you override client-side settings (p. 166), Athena will use them when 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. |
| Step 5: Notify users which workgroups to use for running queries. | Inform your account’s users about workgroup names that they can use. |
| Step 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. 176) and Creating an Amazon SNS Topic in the Amazon Simple Notification Service Getting Started Guide. |
Step | Description
--- | ---
Step 7: Switch to the workgroup so that you can run queries. | To run queries, switch to the appropriate workgroup. For detailed instructions, see "Specify a Workgroup in Which to Run Queries" (p. 171).

IAM Policies for Accessing Workgroups

To control access to workgroups, 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 workgroup policies, see Workgroup Example Policies (p. 162).

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/](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 [IAM User Guide](https://docs.aws.amazon.com/iam/latest/userguide/).
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. 162).
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 the policy to your groups, users, or roles.

   In an identity-based policy, attach the policy to an identity and specify the workgroup resources that a user, a group, or role can access. In a resource-based policy, attach a policy to the workgroup resource that you want to control. In the policy, specify which principals can access that resource.

For more information, see the following topics in the [IAM User Guide](https://docs.aws.amazon.com/iam/latest/userguide/):

- 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. 162).
Workgroup Example Policies

This section includes example policies you can use to enable various actions on workgroups.

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. 161).

- Example Policy for Full Access to All Workgroups (p. 162)
- Example Policy for Full Access to a Specified Workgroup (p. 162)
- Example Policy for Running Queries in a Specified Workgroup (p. 163)
- Example Policy for Running Queries in the Primary Workgroup (p. 164)
- Example Policy for Management Operations on a Specified Workgroup (p. 164)
- Example Policy for Listing Workgroups (p. 165)
- Example Policy for Running and Stopping Queries in a Specific Workgroup (p. 165)
- Example Policy for Working with Named Queries in a Specific Workgroup (p. 165)

Example 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.

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

Example 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.

```
{
    "Version": "2012-10-17",
    "Statement": [ 
        { 
            "Effect": "Allow",
            "Action": [ 
                "athena:*" 
            ],
            "Resource": [ 
                "*"
            ]
        }
    ]
}
```
Example Example Policy for Running Queries in a Specified Workgroup

In the following policy, a user is allowed to run queries in the specified workgroupA, and view them. The user is not allowed to perform management tasks for the workgroup itself, such as updating or deleting it.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "athena:ListWorkGroups",
        "athena:GetExecutionEngine",
        "athena:GetExecutionEngines",
        "athena:GetNamespace",
        "athena:GetCatalogs",
        "athena:GetNamespaces",
        "athena:GetTables",
        "athena:GetTable"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "athena:StartQueryExecution",
        "athena:GetQueryResults",
        "athena:DeleteNamedQuery",
        "athena:GetNamedQuery",
        "athena:ListQueryExecutions",
        "athena:StopQueryExecution",
        "athena:GetQueryResultsStream",
        "athena:GetQueryExecutions",
        "athena:ListNamedQueries",
        "athena:GetNamedQuery",
        "athena:CreateNamedQuery",
        "athena:GetQueryExecution",
        "batchGetNamedQuery",
        "athena:BatchGetQueryExecution"
      ],
      "Resource": [
        "arn:aws:athena:us-east-1:123456789012:workgroup/workgroupA"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "athena:DeleteWorkGroup",
        "athena:UpdateWorkGroup",
        "athena:GetWorkGroup",
        "athena:CreateWorkGroup"
      ],
      "Resource": [
        "arn:aws:athena:us-east-1:123456789012:workgroup/workgroupA"
      ]
    }
  ]
}
```
"Effect": "Allow",
"Action": [
  "athena:ListWorkGroups",
  "athena:GetExecutionEngine",
  "athena:GetExecutionEngines",
  "athena:GetNamespace",
  "athena:GetCatalogs",
  "athena:GetNamespaces",
  "athena:GetTables",
  "athena:GetTable"
],
"Resource": "*"
},
{
"Effect": "Allow",
"Action": [
  "athena:StartQueryExecution",
  "athena:GetQueryResults",
  "athena:DeleteNamedQuery",
  "athena:GetNamedQuery",
  "athena:ListQueryExecutions",
  "athena:StopQueryExecution",
  "athena:GetQueryResultsStream",
  "athena:GetQueryExecutions",
  "athena:ListNamedQueries",
  "athena:GetQueryExecution",
  "athena:BatchGetNamedQuery",
  "athena:BatchGetQueryExecution",
  "athena:GetWorkGroup"
],
"Resource": [
  "arn:aws:athena:us-east-1:123456789012:workgroup/workgroupA"
]
}

**Example Example Policy for Running Queries in the Primary Workgroup**

In the following example, we use the policy that allows a particular user to run queries in the primary workgroup.

**Note**

We recommend that you add this policy to all users who are otherwise configured to run queries in their designated workgroups. Adding this policy 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 the following policy to a resource section of the [Example Policy for Running Queries in a Specified Workgroup (p. 163)](#).  

"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.

{  
  "Effect": "Allow",
  "Action": [ 
    "athena:CreateWorkGroup",
    "athena:DeleteWorkGroup",
    "athena:GetWorkGroup",
    "athena:UpdateWorkGroup"
  ],
  "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:

```json
{  "Effect": "Allow",  "Action": [    "athena:ListWorkGroups"  ],  "Resource": "*"}
```

Example Example Policy for Running and Stopping Queries in a Specific Workgroup

In this policy, a user is allowed to run queries in the workgroup:

```json
```

Example Example Policy for Working with Named Queries in a Specific Workgroup

In the following policy, a user has permissions to create, delete, and obtain information about named queries in the specified workgroup:

```json
```

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.
  • **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. 166). 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. 169).

You can also set query limits (p. 174) 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. 169).

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><strong>Create a Workgroup</strong> (p. 167)</td>
<td>Create a new workgroup.</td>
</tr>
<tr>
<td><strong>Edit a Workgroup</strong> (p. 169)</td>
<td>Edit a workgroup and change its settings. You cannot change a workgroup’s name, but you can create a new workgroup with the same settings and a different name.</td>
</tr>
<tr>
<td><strong>View the Workgroup’s Details</strong> (p. 169)</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 <strong>Override client-side settings</strong> is checked.</td>
</tr>
<tr>
<td><strong>Delete a Workgroup</strong> (p. 170)</td>
<td>Delete a workgroup. If you delete a workgroup, query history, saved queries, the workgroup’s settings and per-query data limit controls are deleted. The workgroup-wide data limit controls remain in CloudWatch, and you can delete them individually. The primary workgroup cannot be deleted.</td>
</tr>
<tr>
<td><strong>Switch between Workgroups</strong> (p. 170)</td>
<td>Switch between workgroups to which you have access.</td>
</tr>
<tr>
<td><strong>Enable and Disable a Workgroup</strong> (p. 171)</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>
</tbody>
</table>

Before you can run queries, you must specify to Athena which workgroup to use. You must have permissions to the workgroup.

Create a Workgroup

Creating a workgroup requires permissions to **CreateWorkgroup** API actions. See Access to Athena Workgroups (p. 59) and IAM Policies for Accessing Workgroups (p. 161). If you are adding tags, you also need to add permissions to **TagResource**. See the section called “Tag Policy Examples” (p. 183).

**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>Workgroup name</td>
<td>Required. Enter a unique name for your workgroup. Use 1 - 128 characters. (A-Z,a-z,0-9,-,.). This name cannot be changed.</td>
</tr>
<tr>
<td>Description</td>
<td>Optional. Enter a description for your workgroup. It can contain up to 1024 characters.</td>
</tr>
<tr>
<td>Query result location</td>
<td>Optional. Enter a path to an Amazon S3 bucket or prefix. This bucket and prefix must exist before you specify them.</td>
</tr>
<tr>
<td>Note</td>
<td>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.</td>
</tr>
<tr>
<td>Encrypt query results</td>
<td>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.</td>
</tr>
<tr>
<td>Publish to CloudWatch</td>
<td>This field is selected by default. Publish query metrics to Amazon CloudWatch. See Viewing Query Metrics (p. 174).</td>
</tr>
<tr>
<td>Override client-side settings</td>
<td>This field is unselected by default. If you select it, workgroup settings apply to all queries in the workgroup and override client-side settings. For more information, see Workgroup Settings Override Client-Side Settings (p. 166).</td>
</tr>
<tr>
<td>Tags</td>
<td>Optional. Add one or more tags to a workgroup. A tag is a label that you assign to an Athena workgroup resource. It consists of a key and a value. Use best practices for AWS tagging strategies to create a consistent set of tags and categorize workgroups by purpose, owner, or environment. You can also use tags in IAM policies, and to control billing costs. Do not use duplicate tag keys the same workgroup. For more information, see Tagging Workgroups (p. 179).</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. 161) 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. 59) and IAM Policies for Accessing Workgroups (p. 161). If you are adding or editing tags, you also need to have permissions to TagResource. See the section called “Tag Policy Examples” (p. 183).

**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. 167). 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:

![Workgroup Overview Example](image)

Delete a Workgroup

You can delete a workgroup if you have permissions to do so. The primary workgroup cannot be 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. 164).

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.

**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. 161).
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. 170).
   - 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:
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. 44).

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. 161).

- 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. 171).

- If you use the API or the drivers to run queries, you must specify the query results location using one of the ways: either for individual queries, using OutputLocation (client-side), or in the workgroup, using WorkGroupConfiguration. If the location is not specified in either way, Athena issues an error at query execution. If you use the Athena console, and don't specify the query results location using one of the methods, Athena uses the default location (p. 84).

- 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 IAM User Guide.

Calls to the BatchGetQueryExecution and BatchGetNamedQuery API operations return information about query executions only for those 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. 172).

- If the workgroup in which a query will run is configured with an enforced query results location (p. 166), 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><code>arn:aws:iam::123456789012:user/abc</code> is not authorized to perform:</td>
<td>A user runs a query in a workgroup, but does not have access to it. Update your policy to have access to the workgroup.</td>
</tr>
<tr>
<td>INVALID_INPUT. WorkGroup &lt;name&gt; is disabled.</td>
<td>A user runs a query in a workgroup, but the workgroup is disabled. Your workgroup could be disabled by your administrator. It is possible also that you don't have access to it. In both cases, contact an administrator who has access to modify workgroups.</td>
</tr>
<tr>
<td>INVALID_INPUT. WorkGroup &lt;name&gt; is not found.</td>
<td>A user runs a query in a workgroup, but the workgroup does not exist. This could happen if the workgroup was deleted. Switch to another workgroup to run your query.</td>
</tr>
<tr>
<td>InvalidRequestException: when calling the StartQueryExecution operation: No output location provided. An output location is required either through the Workgroup result configuration setting or as an API input.</td>
<td>A user runs a query with the API without specifying the location for query results. You must set the output location for query results using one of the two ways: either for individual queries, using OutputLocation (client-side), or in the workgroup, using WorkGroupConfiguration.</td>
</tr>
<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. 166), 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

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. 174). When metrics are published, they are displayed under the **Metrics** tab in the **Workgroups** panel. Metrics are disabled by default for the primary workgroup.

**Topics**

• Enabling CloudWatch Query Metrics (p. 174)
• Monitoring Athena Queries with CloudWatch Metrics (p. 174)
• Setting Data Usage Control Limits (p. 176)

### Enabling CloudWatch Query Metrics

When you create a workgroup in the console, the setting for publishing query metrics to CloudWatch is selected by default.

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:

```json
"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:

• Total amount of data scanned per query (in Megabytes)
• Query state (successful, failed, or canceled)
• Total query execution time (in seconds)
• Query type (DDL or DML)
• Workgroup name

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.

   As a result, the metrics display.

5. Choose the metrics interval that Athena should use to fetch the query metrics from CloudWatch, or choose the refresh icon to refresh the displayed metrics.

To view metrics in the Amazon CloudWatch console

1. Open the Amazon CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. In the navigation pane, choose Metrics.
3. Select the AWS/Athena namespace.

To view metrics with the CLI

• Open a command prompt, and use the following command:

```bash
aws cloudwatch list-metrics --namespace "AWS/Athena"
```

• To list all available metrics, use the following command:

```bash
aws cloudwatch list-metrics --namespace "AWS/Athena"
```

List of CloudWatch Metrics for Athena

If you've enabled CloudWatch metrics in Athena, it sends the following metrics to CloudWatch. The metrics use the AWS/Athena namespace.
## 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. 176).

- **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 choose to take one of the following actions:

- 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. 177). 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.

- 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, stopping any further queries from running.

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.
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:<name>** 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. The **Per Query Data Usage Control** dialog displays.

5. Specify (or update) the field values, as follows:

   - For **Data limit**, specify a value between 10000 KB (minimum) and 7 EB (maximum).

     **Note**
     These are limits imposed by the console for data usage controls within workgroups. They do not represent any query limits in Athena.

   - For units, select the unit value from the drop-down list.

   - Review the default **Action**. The default **Action** is to cancel the query if it exceeds the limit. This action 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:<name> 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 Control 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 10000 KB (minimum) and 7 EB (maximum).
     **Note**
     These are limits imposed by the console for data usage controls within workgroups. They do not represent any query limits in Athena.
   - For units, select the unit value from the drop-down list.
   - For time period, choose a time period from the drop-down list.
   - For Action, choose the Amazon SNS topic from the drop-down list, if you have it 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.
Tagging Workgroups

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 use tags to categorize your AWS resources in different ways; for example, by purpose, owner, or environment. For Athena, the workgroup is the resource that you can tag. For example, you can create a set of tags for workgroups in your account that helps you track workgroup owners, or identify workgroups by their purpose. We recommend that you use AWS tagging best practices to create a consistent set of tags to meet your organization requirements.

You can work with tags using the Athena console or the API operations.

Topics
- Tag Basics (p. 179)
- Tag Restrictions (p. 180)
- Working with Tags Using the Console (p. 180)
- Working with Tags Using the API Actions (p. 182)
- Tag-Based IAM Access Control Policies (p. 183)

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 you can add, edit, or remove tags from an existing workgroup. You can edit a tag in the console. If you use the API operations, to edit a tag, remove the old tag and add a new one. If you delete a workgroup, any tags for it are also deleted. Other workgroups in your account continue using the same tags.

Athena does not automatically assign tags to your resources, such as your workgroups. You can edit tag keys and values, and you can remove tags from a workgroup 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 at the same time to the same workgroup. If you do, Athena issues an error message. If you tag a workgroup using an existing tag key in a separate TagResource action, the new tag value overwrites the old value.

In IAM, you can control which users in your AWS account have permission to create, edit, remove, or list tags. For more information, see the section called “Tag Policy Examples” (p. 183).

For a complete list of Amazon Athena tag actions, see the API action names in the Amazon Athena API Reference.

You can use the same 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. 180).
Tag Restrictions

Tags have the following restrictions:

• In Athena, you can tag workgroups. You cannot tag queries.
• Maximum number of tags per workgroup is 50. To stay within the limit, review and delete unused tags.
• For each workgroup, 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 workgroup. If you do, Athena issues an error message. If you tag a workgroup 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.
• Don’t use the "aws:" prefix in tag keys; it’s 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 AWS account.

Working with Tags Using 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. 180)
• Adding and Deleting Tags on an Individual Workgroup (p. 180)

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 choose Manage Tags. 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.
To add a tag when creating a new workgroup

**Note**
Make sure you give a user IAM permissions to the TagResource and CreateWorkGroup actions if you want to allow them to add tags when creating a workgroup in the console, or pass in tags upon CreateWorkGroup.

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. 167).
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. 180).
5. When you are done, choose Create Workgroup.

To add or edit a tag to an existing workgroup

**Note**
Make sure you give a user IAM permissions to the TagResource and CreateWorkGroup actions if you want to allow them to add tags when creating a workgroup in the console, or pass in tags upon CreateWorkGroup.

1. Open the Athena console at https://console.aws.amazon.com/athena/, choose the Workgroups tab, and select the workgroup.
2. Choose View details or Edit workgroup.
3. Choose the Tags tab.
4. On the Tags tab, choose Manage tags, and then specify the key and value for each tag. For more information, see Tag Restrictions (p. 180).
5. When you are done, 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, choose View details, and then choose the Tags tab.
3. On the Tags tab, choose Manage tags.
4. In the list of tags, select the **delete** button (a cross) for the tag, and choose **Save**.

## Working with Tags Using the API Actions

You can also use the `CreateWorkGroup` API operation with the optional tag parameter that you can use to pass in one or more tags for the workgroup. To add, remove, or list tags, you can use the following AWS API operations: `TagResource`, `UntagResource`, and `ListTagsForResource`.

### Tags API Actions in Athena

<table>
<thead>
<tr>
<th>API name</th>
<th>Action description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TagResource</td>
<td>Add or overwrite one or more tags to the workgroup with the specified ARN.</td>
</tr>
<tr>
<td>UntagResource</td>
<td>Delete one or more tags from the workgroup with the specified ARN.</td>
</tr>
<tr>
<td>ListTagsForResource</td>
<td>List one or more tags for the workgroup resource with the specified ARN.</td>
</tr>
</tbody>
</table>

For more information, see the Amazon Athena API Reference.

### Example TagResource

In the following example, we add two tags to `workgroupA`:

```java
List<Tag> tags = new ArrayList<>();
tags.add(new Tag().withKey("tagKey1").withValue("tagValue1"));
tags.add(new Tag().withKey("tagKey2").withValue("tagValue2"));

TagResourceRequest request = new TagResourceRequest()
    .withTags(tags);

client.tagResource(request);
```

**Note**

Do not add duplicate tag keys at the same time to the same workgroup. If you do, Athena issues an error message. If you tag a workgroup using an existing tag key in a separate `TagResource` action, the new tag value overwrites the old value.

### Example UntagResource

In the following example, we remove `tagKey2` from `workgroupA`:

```java
List<String> tagKeys = new ArrayList<>();
tagKeys.add("tagKey2");

UntagResourceRequest request = new UntagResourceRequest()
    .withTagKeys(tagKeys);

client.untagResource(request);
```

### Example ListTagsForResource

In the following example, we list tags for `workgroupA`:

```java
```
Tag-Based IAM Access Control Policies

Having tags allows you to write an IAM policy that includes the Condition block to control access to workgroups based on their tags.

Tag Policy Examples

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:GetExecutionEngine",
                "athena:GetExecutionEngines",
                "athena:GetNamespace",
                "athena:GetCatalogs",
                "athena:GetNamespaces",
                "athena:GetTables",
                "athena:GetTable"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "athena:StartQueryExecution",
                "athena:GetQueryResults",
                "athena:DeleteNamedQuery",
                "athena:GetNamedQuery",
                "athena:ListQueryExecutions",
                "athena:StopQueryExecution",
                "athena:GetQueryResultsStream",
                "athena:GetQueryExecutions",
                "athena:GetNamedQuery",
                "athena:ListNamedQuery",
                "athena:CreateNamedQuery",
                "athena:GetNamedQuery",
                "athena:GetQueryExecution",
                "athena:BatchGetNamedQuery",
                "athena:BatchGetQueryExecution",
                "athena:GetWorkGroup",
                "athena:TagResource",
                "athena:UntagResource",
                "athena:ListTagsForResource"
            ],
        }
    ]
}
```
Example 2: Policy Block that Denies Actions on a Workgroup Based on a Tag Key and Tag Value Pair

Tags that are associated with an existing workgroup are referred to as resource tags. Resource tags let you write policy blocks, such as the following, which deny the listed actions on any workgroup tagged with tag key and tag value pair, such as: stack, production.

```
{
  "Effect": "Deny",
  "Action": [
    "athena:StartQueryExecution",
    "athena:GetQueryResults",
    "athena:DeleteNamedQuery",
    "athena:UpdateWorkGroup",
    "athena:GetNamedQuery",
    "athena:ListQueryExecutions",
    "athena:GetWorkGroup",
    "athena:StopQueryExecution",
    "athena:GetQueryResultsStream",
    "athena:GetQueryExecutions",
    "athena:ListNamedQueries",
    "athena:CreateNamedQuery",
    "athena:GetQueryExecution",
    "athena:BatchGetNamedQuery",
    "athena:BatchGetQueryExecution",
    "athena:TagResource",
    "athena:UntagResource",
    "athena:ListTagsForResource"
  ],
  "Resource": "arn:aws:athena:us-east-1:123456789012:workgroup/*",
  "Condition": {
    "StringEquals": {
      "aws:ResourceTag/stack": "production"
    }
  }
}
```

Example 3. Policy Block that Restricts Tag-Changing Action Requests to Specified Tags

Tags passed in as parameters to a tag-mutating API action, such as CreateWorkGroup with tags, TagResource, and UntagResource, are referred to as request tags. Use these tags, as shown in the following example policy block. This allows CreateWorkGroup only if one of the tags included when you create a workgroup is a tag with the costcenter key with one of the allowed tag values: 1, 2, or 3.

Note: Make sure that you give a user IAM permissions to the TagResource and CreateWorkGroup API operations, if you want to allow them to pass in tags upon CreateWorkGroup.

```
{
  "Effect": "Allow",
  "Action": [
    "athena:CreateWorkGroup",
    "athena:TagResource"
  ],
  "Resource": "arn:aws:athena:us-east-1:123456789012:workgroup/**",
  "Condition": {
    "StringEquals": {
      "aws:RequestTag/costcenter": [
        "1",
        "2",
```
"3"

};

} }
Monitoring Logs and Troubleshooting

Examine Athena requests using CloudTrail logs and troubleshoot queries.

Topics
- Logging Amazon Athena API Calls with AWS CloudTrail (p. 186)
- Troubleshooting (p. 189)

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 also use Athena to query CloudTrail log files for insight. For more information, see Querying AWS CloudTrail Logs (p. 135) and CloudTrail SerDe (p. 195).

Athena Information in CloudTrail

CloudTrail is enabled on your AWS 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 AWS account. For more information, see Viewing Events with CloudTrail Event History.

For an ongoing record of events in your AWS 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. 187)
- StartQueryExecution (Failed) (p. 188)
- CreateNamedQuery (p. 188)

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/"
        },
        "query":"Select 10"},
    "responseElements":{
        "queryExecutionId":"b621c254-74e0-48e3-9630-78ed857782f9"
    },
    "requestID":"f5039b01-305f-11e7-b146-c3fc56a7dc7a",
    "eventID":"c97cf8c8-6112-467a-8777-53bb38f83fd5",
```

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": "2017-05-04T00:21:57Z",
  "eventSource": "athena.amazonaws.com",
  "eventName": "StartQueryExecution",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "77.88.999.69",
  "userAgent": "aws-internal/3",
  "errorCode": "InvalidRequestException",
  "errorMessage": "Invalid result configuration. Should specify either output location or result configuration",
  "requestParameters": {
    "clientRequestToken": "ca0e965f-d6d8-4277-8257-814a57f57446",
    "query": "Select 10"
  },
  "responseElements": null,
  "requestID": "aefbc057-305f-11e7-9f39-bbc56d5d161e",
  "eventID": "6e1fc69b-d076-477e-8dec-024ee51488c4",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```

CreateNamedQuery

```
{
  "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": "2017-05-16T22:00:58Z",
  "eventSource": "athena.amazonaws.com",
  "eventName": "CreateNamedQuery",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "77.88.999.69",
  "userAgent": "aws-cli/1.11.85 Python/2.7.10 Darwin/16.6.0 botocore/1.5.48",
  "requestParameters": {
    "name": "johndoetest",
    "queryString": "select 10",
    "database": "default",
    "clientRequestToken": "fc1ad880-69ee-4df0-bb0f-1770d9a539b1"
  },
  "responseElements": {
    "namedQueryId": "cdd0fe29-4787-4263-9188-a9c8db29f2d6"
  }
}
```
Troubleshooting

Use these documentation topics to troubleshoot problems with Amazon Athena.

- Service Limits (p. 253)
- Limitations (p. 238)
- Unsupported DDL (p. 237)
- Names for Tables, Databases, and Columns (p. 71)
- Data Types (p. 216)
- Supported SerDes and Data Formats (p. 191)
- Compression Formats (p. 215)
- Reserved Words (p. 72)
- Troubleshooting Workgroups (p. 172)

In addition, use the following AWS resources:

- Athena topics in the AWS Knowledge Center
- Athena discussion forum
- Athena posts in the AWS Big Data Blog
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. 190)
- Supported SerDes and Data Formats (p. 191)
- Compression Formats (p. 215)

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:

- Use DDL statements to describe how to read and write data to the table and do not specify a `ROW FORMAT`, as in this example. This omits listing the actual SerDe type and the native `LazySimpleSerDe` is used by default.

In general, Athena uses the `LazySimpleSerDe` if you do not specify a `ROW FORMAT`, or if you specify `ROW FORMAT DELIMITED`.

```
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',',
ESCAPED BY '\'
COLLECTION ITEMS TERMINATED BY '|' 
MAP KEYS TERMINATED BY ':'
```

- Explicitly specify the type of SerDe Athena should use when it reads and writes data to the table. Also, specify additional properties in `SERDEPROPERTIES`, as in this example.

```
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'
WITH SERDEPROPERTIES (
  'serialization.format' = ',',
  'field.delim' = ',',
  'collection.delim' = '|',
  'mapkey.delim' = ':',
  'escape.delim' = '\'
)
```
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. 91).

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. 205) if your data does not include values enclosed in quotes.  
  • Use the OpenCSVSerDe for Processing CSV (p. 197) when your data includes quotes in values, or different separator or escape characters. |
| TSV (Tab-Separated Values)      | For data in TSV, each line represents a data record, and each record consists of one or more fields, separated by tabs. | Use the LazySimpleSerDe for CSV, TSV, and Custom-Delimited Files (p. 205) and specify the separator character as FIELDS TERMINATED BY '\t'. |
| Custom-Delimited                | For data in this format, each line represents a data record, and records are separated by custom delimiters. | Use the LazySimpleSerDe for CSV, TSV, and Custom-Delimited Files (p. 205) and specify custom delimiters. |
| 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. 202).  
  • Use the OpenX JSON SerDe (p. 203). |
| Apache Avro                     | A format for storing data in Hadoop that uses JSON-based schemas for record values. | Use the Avro SerDe (p. 192). |
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. In ROW FORMAT, specify the Avro SerDe as follows: ROW
FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'. In SERDEPROPERTIES, specify the schema, as shown in the following example.

**Note**
You can query data in regions other than the region where you run Athena. Standard inter-
region data transfer rates for Amazon S3 apply in addition to standard Athena charges. To
reduce data transfer charges, replace myregion in s3://athena-examples-myregion/
path/to/data/ with the region identifier where you run Athena, for example, s3://athena-
examples-us-east-1/path/to/data/.

```
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",  
```

193
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.

**RegexSerDe for Processing Apache Web Server Logs**

**SerDe Name**

RegexSerDe

**Library Name**

RegexSerDe

**Examples**

The following example creates a table from CloudFront logs using the RegExSerDe from the Getting Started tutorial.
Note
You can query data in regions other than the region where you run Athena. Standard inter-region data transfer rates for Amazon S3 apply in addition to standard Athena charges. To reduce data transfer charges, replace myregion in s3://athena-examples-myregion/path/to/data/ with the region identifier where you run Athena, for example, s3://athena-examples-us-east-1/path/to/data/.

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. 117).

```sql
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,
      accountId:STRING,
      type:STRING>>,
    eventType STRING,
    apiVersion STRING,
    readOnly STRING,
    recipientAccountId STRING,
    serviceEventDetails STRING,
    sharedEventID STRING,
    vpcEndpointId 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/';
```

The following query returns the logins that occurred over a 24-hour period:

```sql
SELECT
  ...
```
OpenCSV SerDe for Processing CSV

When you create a table from CSV data in Athena, determine what types of values it contains:

- If data contains values enclosed in double quotes ("), you can use the OpenCSV SerDe to deserialize the values in Athena. In the following sections, note the behavior of this SerDe with STRING data types.
- If 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. 205).

CSV SerDe (OpenCSV SerDe)

The OpenCSV SerDe behaves as follows:

- Converts all column type values to STRING.
- To recognize data types other than STRING, relies on the Presto parser and converts the values from STRING into those data types if it can recognize them.
- Uses double quotes ("), as the default quote character, and allows you to specify separator, quote, and escape characters, such as:

  ```
  WITH SERDEPROPERTIES ("separatorChar" = ",", "quoteChar" = "\", "escapeChar" = "\\")
  ```

  - Cannot escape \t or \n directly. To escape them, use "escapeChar" = "\\". See the example in this topic.
  - Does not support embedded line breaks in CSV files.

**Note**

When you use Athena with OpenCSVSerDe, the SerDe converts all column types to STRING. Next, the parser in Athena parses the values from STRING into actual types based on what it finds. For example, it parses the values into BOOLEAN, BIGINT, INT, and DOUBLE data types when it can discern them. If the values are in TIMESTAMP in the UNIX format, Athena parses them as TIMESTAMP. If the values are in TIMESTAMP in Hive format, Athena parses them as INT. DATE type values are also parsed as INT.

To further convert columns to the desired type in a table, you can create a view (p. 86) over the table and use CAST to convert to the desired type.

For data types other than STRING, when the parser in Athena can recognize them, this SerDe behaves as follows:

- Recognizes BOOLEAN, BIGINT, INT, and DOUBLE data types and parses them without changes.
- Recognizes the TIMESTAMP type if it is specified in the UNIX format, such as yyyy-mm-dd hh:mm:ss[.f...], as the type LONG.
• Does not support `TIMESTAMP` in the JDBC-compliant `java.sql.Timestamp` format, such as "YYYY-MM-DD HH:MM:SS.fffffff" (9 decimal place precision). If you are processing CSV data from Hive, use the UNIX format for `TIMESTAMP`.

• Recognizes the `DATE` type if it is specified in the UNIX format, such as `YYYY-MM-DD`, as the type `LONG`.

• Does not support `DATE` in another format. If you are processing CSV data from Hive, use the UNIX format for `DATE`.

Example: Escaping `\t` or `\n`

Consider the following test data:

```
"  123 \t\n  ",abc
" 456 ",xyz
```

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" = ",", "quoteChar" = "\", "escapeChar" = "\\")
LOCATION 's3://user-test-region/dataset/test1/'
```

Next, run the following query:

```
select * from test1;
```

It returns this result, correctly escaping `\t` or `\n`:

```
f1      s2
\t\t\n123 \t\n\nabc
456     xyz
```

SerDe Name

CSV SerDe

Library Name

To use this SerDe, specify its fully qualified class name in `ROW FORMAT`. Also specify the delimiters inside `SERDEPROPERTIES`, as follows:

```
... 
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (  
  "separatorChar" = ",",  
  "quoteChar" = "\",  
  "escapeChar" = "\\"
)
```

Example

This example presumes data in CSV saved in `s3://mybucket/mycsv/` with the following contents:
Use a `CREATE TABLE` statement to create an Athena table based on the data, and reference the `OpenCSVSerDe` class in `ROW FORMAT`, also specifying SerDe properties for character separator, quote character, and escape character, as follows:

```
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://location/of/csv/';
```

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
```

**Note**
The flight table data comes from Flights provided by US Department of Transportation, Bureau of Transportation Statistics. Desaturated from original.

**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 predefined 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 ([^"\"]*)\nINSIDE_BRACKETS ([^\]\])'*`. 

---

199
Amazon Athena User Guide
Grok SerDe

- 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.

```
CREATE EXTERNAL TABLE `mygroktable`(
  syslogbase string,
  queue_id string,
  syslog_message string
) ROW FORMAT SERDE
'com.amazonaws.glue.serde.GrokSerDe'
WITH SERDEPROPERTIES (
  'input.grokCustomPatterns' = 'POSTFIX_QUEUEID [0-9A-F]{7,12}',
  'input.format'='%{SYSLOGBASE} %{POSTFIX_QUEUEID:queue_id}: %{GREEDYDATA:syslog_message}'
) STORED AS INPUTFORMAT
'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION
's3://mybucket/groksample';
```

Start with a simple pattern, such as `%{NOTSPACE:column}`, to get the columns mapped first and then specialize the columns if needed.

Example 2

In the following example, you create a query for Log4j logs. The example logs have the entries in this format:

```
2017-09-12 12:10:34,972 INFO  - processType=AZ, processId=ABCDEFG614B6F5E49, status=RUN,
threadId=123:amqListenerContainerPool123[P:AJ|ABCDE9614B6F5E49||
2017-09-12T12:10:11.172-0700],
executionTime=7290, tenantId=12456, userId=123123f8535f8d76015374e7a1d87c3c,
shard=testapp1,
jobId=12312345e7df0015e777fb2e03f3c, 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, status STRING, threadid STRING, executiontime INT, tenantid INT, userid STRING, shard STRING, jobid STRING, messagetype STRING, action STRING, hostname STRING)
ROW FORMAT SERDE 'com.amazonaws.glue.serde.GrokSerDe'

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
JSON SerDe Libraries

In Athena, you can use two SerDe libraries for processing data in JSON:

- The native Hive JSON SerDe (p. 202)
- The OpenX JSON SerDe (p. 203)

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 used to process JSON data, most commonly events. These events are represented as blocks of JSON-encoded text separated by a new line.

You can also use the Hive JSON SerDe to parse more complex JSON-encoded data with nested structures. However, this requires having a matching DDL representing the complex data types. See Example: Deserializing Nested JSON (p. 204).

With this SerDe, duplicate keys are not allowed in map (or struct) key names.

Note

You can query data in regions other than the region where you run Athena. Standard inter-region data transfer rates for Amazon S3 apply in addition to standard Athena charges. To reduce data transfer charges, replace myregion in s3://athena-examples-myregion/path/to/data/ with the region identifier where you run Athena, for example, s3://athena-examples-us-east-1/path/to/data/.

The following DDL statement uses the Hive JSON SerDe:
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';

**OpenX JSON SerDe**

The OpenX SerDe is used by Athena for deserializing data, which means converting it from the JSON format in preparation for serializing it to the Parquet or ORC format. This is one of two deserializers that you can choose, depending on which one offers the functionality that you need. The other option is the Hive JsonSerDe.

This SerDe has a few useful properties that you can specify when creating tables in Athena, to help address inconsistencies in the data:

**ignore.malformed.json**

Optional. When set to TRUE, lets you skip malformed JSON syntax. The default is FALSE.

**ConvertDotsInJsonKeysToUnderscores**

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. By default, Athena requires that all keys in your JSON dataset use lowercase. The default is TRUE. When set to TRUE, the SerDe converts all uppercase columns to lowercase. Using WITH SERDE PROPERTIES ("case.insensitive"= FALSE;) allows you to use case-sensitive key names in your data.

**ColumnToJsonKeyMappings**

Optional. Maps column names to JSON keys that aren’t identical to the column names. This is useful when the JSON data contains keys that are keywords (p. 72). For example, if you have a JSON key named timestamp, set this parameter to ("ts": "timestamp") to map this key to a column named ts. This parameter takes values of type string. It uses the following key pattern: ^(?!s*$).+
With this SerDe, duplicate keys are not allowed in map (or struct) key names.

The following DDL statement uses the OpenX JSON SerDe:

```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

JSON data can be challenging to deserialize when creating a table in Athena.

When dealing with complex nested JSON, there are common issues you may encounter. For more information about these issues and troubleshooting practices, see the AWS Knowledge Center Article I receive errors when I try to read JSON data in Amazon Athena.

For more information about common scenarios and query tips, see Create Tables in Amazon Athena from Nested JSON and Mappings Using JSONSerDe.

The following example demonstrates a simple approach to creating an Athena table from data with nested structures in JSON. To parse JSON-encoded data in Athena, each JSON document must be on its own line, separated by a new line.

This example presumes a JSON-encoded data with 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"
    }
  
```
The following `CREATE TABLE` command uses the `Openx-JsonSerDe` with collection data types like `struct` and `array` 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` and map key names. Duplicate keys are not allowed in map (or `struct`) 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/';
```

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.

**Library Name**

The Class library name for the LazySimpleSerDe is `org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe`. For more information, see [LazySimpleSerDe](#).

**Examples**

The following examples show how to create tables in Athena from CSV and TSV, using the LazySimpleSerDe. To deserialize custom-delimited file using this SerDe, specify the delimiters similar to the following examples.

- **CSV Example (p. 206)**
• TSV Example (p. 208)

Note
You can query data in regions other than the region where you run Athena. Standard inter-region data transfer rates for Amazon S3 apply in addition to standard Athena charges. To reduce data transfer charges, replace `myregion` in `s3://athena-examples-myregion/path/to/data/` with the region identifier where you run Athena, for example, `s3://athena-examples-us-east-1/path/to/data/`.

Note
The flight table data comes from Flights provided by US Department of Transportation, Bureau of Transportation Statistics. Desaturated from original.

CSV Example

Use the `CREATE TABLE` statement to create an Athena table from the underlying data in CSV stored in Amazon S3.

```
CREATE EXTERNAL TABLE flight_delays_csv (  
    yr INT,  
    quarter INT,  
    month INT,  
    dayofmonth INT,  
    dayofweek INT,  
    flightdate STRING,  
    uniquecarrier STRING,  
    airlineid INT,  
    carrier STRING,  
    tailnum STRING,  
    flightnum STRING,  
    originairportid INT,  
    originairportseqid INT,  
    origincitymarketid INT,  
    origin STRING,  
    origininame STRING,  
    originstate STRING,  
    originstatefips STRING,  
    originwac INT,  
    destairportid INT,  
    destairportseqid INT,  
    destcitymarketid INT,  
    dest STRING,  
    destiname STRING,  
    deststate STRING,  
    deststatefips STRING,  
    destwac INT,  
    crsdeptime STRING,  
    deptime STRING,  
    depdelay INT,  
    depdelayminutes INT,  
    depdel15 INT,  
    departuredelaygroups INT,  
    deptimeblk STRING,  
    taxiout INT,  
    wheelsoff STRING,  
    wheelson STRING,  
    taxin INT,  
    crsarrtime INT,  
    arrtime STRING,  
    arrdelay INT,  
```
arrdelayminutes INT,
arrdel15 INT,
arrivaldelaygroups INT,
arrtimeblk STRING,
cancelled INT,
cancellationcode STRING,
diverted INT,
crselapsedtime INT,
actualelapsedtime INT,
airtime INT,
arrtime INT,
distance INT,
distancegroup INT,
carrildelay INT,
weatherdelay INT,
nasdelay INT,
securitydelay INT,
lateaircraftdelay INT,
firstdeptime STRING,
totaddgtime INT,
longestaddgtime INT,
divairportlandings INT,
divreacheddest INT,
divactualelapsedtime INT,
divarrdelay INT,
divdistance INT,
div1airport STRING,
div1airportid INT,
div1airportsegid INT,
div1wheelsonl STRING,
div1totalgtme INT,
div1longestgtme INT,
div1wheelsoff STRING,
div1tailnum STRING,
div2airport STRING,
div2airportid INT,
div2airportsegid INT,
div2wheelsonl STRING,
div2totalgtme INT,
div2longestgtme INT,
div2wheelsoff STRING,
div2tailnum STRING,
div3airport STRING,
div3airportid INT,
div3airportsegid INT,
div3wheelsonl STRING,
div3totalgtme INT,
div3longestgtme INT,
div3wheelsoff STRING,
div3tailnum STRING,
div4airport STRING,
div4airportid INT,
div4airportsegid INT,
div4wheelsonl STRING,
div4totalgtme INT,
div4longestgtme INT,
div4wheelsoff STRING,
div4tailnum STRING,
div5airport STRING,
div5airportid INT,
div5airportsegid INT,
div5wheelsonl STRING,
div5totalgtme INT,
div5longestgtme 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:

```sql
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**

This example presumes source data in TSV saved in s3://mybucket/mytsv/.

Use a CREATE TABLE statement to create an Athena table from the TSV data stored in Amazon S3. Notice that this example does not reference any SerDe class in ROW FORMAT because it uses the LazySimpleSerDe, and it can be omitted. The example specifies SerDe properties for character and line separators, and an escape character:

```sql
CREATE EXTERNAL TABLE flight_delays_tsv (yr INT, quarter INT, month INT, dayofmonth INT, dayofweek INT, flightdate STRING, uniquecarrier STRING, airlineid INT, carrier STRING, tailnum STRING, flightnum STRING, originairportid INT, originairportseqid INT, origincitymarketid INT, origin STRING, origincityname STRING, originstate STRING, originstatefips STRING, originwac INT, destairportid INT, destairportseqid INT, destcitymarketid INT, dest STRING, destcityname STRING, deststate STRING, deststatefips STRING, deststatename STRING, destwac INT,
```
crsdeptime STRING,
depdelay INT,
depdel15 INT,
departuredelaygroups INT,
departuretime STRING,
taxiout INT,
wheelsoff STRING,
wheelseon STRING,
taxiin INT,
crsarrtime INT,
arrtime STRING,
adelay INT,
adelayminutes INT,
adeld15 INT,
arrivaldelaygroups INT,
arrtimeblock STRING,
cancelled INT,
cancellationcode STRING,
diverted INT,
crselapsedtime INT,
actualelapsedtime INT,
airtime INT,
airlines INT,
distance INT,
distancegroup INT,
carrier INT,
weather INT,
nas INT,
security INT,
lateaircraft INT,
firstdeptime STRING,
totaladgtime INT,
longestadgtime INT,
divairportlandings INT,
divelreacheddest INT,
divelactualelapsedtime INT,
diveldelay INT,
diveldistance INT,
divelairport STRING,
divelairportid INT,
divelairportseqid INT,
divelwheelsoff STRING,
diveltotaltime INT,
divellongesttime INT,
divelwheelsoff STRING,
diveltailnumber STRING,
divelairport STRING,
divel2airport INT,
divel2airportseqid INT,
divel2wheelsoff STRING,
divel2totaltime INT,
divel2longesttime INT,
divel2wheelsoff STRING,
divel2tailnumber STRING,
divel3airport STRING,
divel3airportid INT,
divel3airportseqid INT,
divel3wheelsoff STRING,
divel3totaltime INT,
divel3longesttime INT,
divel3wheelsoff STRING,
divel3tailnumber STRING,
divel4airport STRING,
divel4airportid INT,
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**
You can query data in regions other than the region where you run Athena. Standard inter-region data transfer rates for Amazon S3 apply in addition to standard Athena charges. To reduce data transfer charges, replace `myregion` in `s3://athena-examples-myregion/`
path/to/data/ with the region identifier where you run Athena, for example, s3://athena-examples-us-east-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,  
  originalairportseqid INT,  
  originairportname STRING,  
  origin STRING,  
  originstate STRING,  
  originstatefips STRING,  
  originwac INT,  
  destairportid INT,  
  destairportseqid INT,  
  destairportname STRING,  
  dest STRING,  
  deststate STRING,  
  deststatename STRING,  
  destwac INT,  
  crsdeptime STRING,  
  deptime STRING,  
 .dep delay INT,  
  depdelayminutes INT,  
  depdells INT,  
  departuredelaygroups INT,  
  deptimeblk STRING,  
  taxiout INT,  
  wheels off STRING,  
  wheelson STRING,  
  taxiin INT,  
  crsarrtime INT,  
  arrtime STRING,  
  arrdelay INT,  
  arrdelayminutes INT,  
  arrdells INT,  
  arrivaldelaygroups INT,  
  arrtimeblk STRING,  
  cancelled INT,  
  cancellationcode STRING,  
  diverted INT,  
  crselapsedtime INT,  
  actualelapsedtime INT,  
  airt ime INT,  
  flights INT,  
  distance INT,  
  distancegroup INT,  
  carrierdelay INT,  
  weath erdelay INT,
```
nasdelay INT,
securitydelay INT,
lateaircraftdelay INT,
firstdeptime STRING,
totaladdgtime INT,
longestaddgtime INT,
divairportlandings INT,
divreacheddest INT,
divactualelapsedtime 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)
STORED AS ORC
LOCATION 's3://athena-examples-myregion/flight/orc/'
tblproperties ("orc.compress"="ZLIB");

Run the `MSCK REPAIR TABLE` statement on the table to refresh partition metadata:

MSCK REPAIR TABLE flight_delays_orc;

Use this query to obtain the top 10 routes delayed by more than 1 hour:

SELECT origin, dest, count(*) as delays
FROM flight_delays_orc
WHERE depdelayminutes > 60
GROUP BY origin, dest
ORDER BY 3 DESC
LIMIT 10;

**Parquet SerDe**

**SerDe Name**

ParquetHiveSerDe is used for data stored in Parquet Format.

**Library Name**

Athena uses this class when it needs to deserialize data stored in Parquet: org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe.

**Example: Querying a File Stored in Parquet**

*Note*

You can query data in regions other than the region where you run Athena. Standard inter-region data transfer rates for Amazon S3 apply in addition to standard Athena charges. To reduce data transfer charges, replace `myregion` in `s3://athena-examples-myregion/path/to/data/` with the region identifier where you run Athena, for example, `s3://athena-examples-us-east-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:

```sql
CREATE EXTERNAL TABLE flight_delays_pq (
  yr INT,
  quarter INT,
  month INT,
  dayofmonth INT,
  dayofweek INT,
  flightdate STRING,
  uniquecarrier STRING,
  airlineid INT,
  carrier STRING,
  tailnum STRING,
  flightnum STRING,
  originairportid INT,
  originairportseqid INT,
  origincitymarketid INT,
  origin STRING,
  origincityname STRING,
  originstate STRING,
  originstatefips STRING,
  originstatename STRING,
  originwac INT,
  destairportid INT,
  destairportseqid INT,
  destcitymarketid INT,
  dest STRING,
  destcityname STRING,
  deststate STRING,
  deststatefips STRING,
  deststatename STRING,
  destwac INT,
  crsdeptime STRING,
```
deptime STRING,
depdelay INT,
depdelayminutes INT,
depdel15 INT,
departuredelaygroups INT,
deptimeblk STRING,
taxiout INT,
wheelsoff STRING,
wheelson STRING,
taxiin INT,
crsarrtime INT,
arrt ime STRING,
arrdelay INT,
arrdelayminutes INT,
arrdel15 INT,
arrivaldelaygroups INT,
arrt imeblk 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,
divar delay INT,
divdistance INT,
divairport STRING,
divairportid INT,
divairportseqid INT,
divwheelsoff STRING,
divtotaltime INT,
divlongesttime INT,
divtailnum STRING,
div1airport STRING,
div2airport STRING,
divairportid INT,
divairportseqid INT,
divwheelsoff STRING,
div1totaltime INT,
div1longesttime INT,
div1tailnum STRING,
div1wheelsoff STRING,
div1wheelsoff STRING,
div1wheelsoff STRING,
div2airportid INT,
div2airportseqid INT,
div2wheelsoff STRING,
div2wheelsoff STRING,
div2wheelsoff STRING,
div3airportid INT,
div3airportseqid INT,
div3wheelsoff STRING,
div3wheelsoff STRING,
div3wheelsoff STRING,
div4airportid INT,
div4airportseqid INT,
Compression Formats

Athena supports the following compression formats:

- **SNAPPY.** This is the default compression format for files in the Parquet data storage format.
- **ZLIB.** This is the default compression format for files in the ORC data storage format.
- **LZO**
- **GZIP.**

For data in CSV, TSV, and JSON, Athena determines the compression type from the file extension. If it is not present, the data is not decompressed. If your data is compressed, make sure the file name includes the compression extension, such as gz.

Use the GZIP compression in Athena for querying Amazon Kinesis Data Firehose logs. Athena and Amazon Kinesis Data Firehose each support different versions of SNAPPY, so GZIP is the only compatible format.

---

Note
The flight table data comes from Flights provided by US Department of Transportation, Bureau of Transportation Statistics. Desaturated from original.
DDL and SQL Reference

Athena supports a subset of DDL statements and ANSI SQL functions and operators.

Topics

• Data Types (p. 216)
• DDL Statements (p. 217)
• SQL Queries, Functions, and Operators (p. 233)
• Unsupported DDL (p. 237)
• Limitations (p. 238)

Data Types

When you run CREATE TABLE, you must specify column names and their data types. For a complete syntax of this command, see CREATE TABLE (p. 222).

List of Supported Data Types in Athena

Athena supports the following data types:

• BOOLEAN. Values are true and false.
• Integer types
  • 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.
• Floating-point types
  • DOUBLE
  • FLOAT
• Fixed precision type
  • 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'.

• String types
  • 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**.

• **BINARY** (for data in Parquet)

• **Date and time types**
  - **DATE**, in the UNIX format, such as `YYYY-MM-DD`.
  - **TIMESTAMP**. Instant in time and date in the UNIX format, such as `yyyy-mm-dd hh:mm:ss[.f...]`. For example, `TIMESTAMP '2008-09-15 03:04:05.324'`. This format uses the session time zone.

• **Structural types**
  - **ARRAY** `< data_type >`
  - **MAP** `< primitive_type, data_type >`
  - **STRUCT** `< col_name : data_type [COMMENT col_comment] [, ...] >`

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.

## DDL Statements

Use the following DDL statements directly in Athena.

Athena query engine is based on **Hive DDL**.

Athena does not support all DDL statements. For information, see **Unsupported DDL (p. 237)**.

### Topics

- **ALTER DATABASE SET DBPROPERTIES** (p. 218)
- **ALTER TABLE ADD PARTITION** (p. 218)
- **ALTER TABLE DROP PARTITION** (p. 219)
- **ALTER TABLE RENAME PARTITION** (p. 220)
- **ALTER TABLE SET LOCATION** (p. 220)
- **ALTER TABLE SET TBLPROPERTIES** (p. 221)
- **CREATE DATABASE** (p. 221)
- **CREATE TABLE** (p. 222)
- **CREATE TABLE AS** (p. 225)
- **CREATE VIEW** (p. 227)
- **DESCRIBE TABLE** (p. 227)
- **DESCRIBE VIEW** (p. 228)
- **DROP DATABASE** (p. 228)
- **DROP TABLE** (p. 229)
- **DROP VIEW** (p. 229)
- **MSCK REPAIR TABLE** (p. 230)
- **SHOW COLUMNS** (p. 230)
- **SHOW CREATE TABLE** (p. 230)
- **SHOW CREATE VIEW** (p. 231)
- **SHOW DATABASES** (p. 231)
- **SHOW PARTITIONS** (p. 231)
- **SHOW TABLES** (p. 232)
• SHOW TBLPROPERTIES (p. 232)
• SHOW VIEWS (p. 233)

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 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. 74).

Synopsis

ALTER TABLE table_name ADD [IF NOT EXISTS]
PARTITION
(partition_col1_name = partition_col1_value
[, partition_col2_name = partition_col2_value]
[,...])
[LOCATION 'location1']
[PARTITION
(partition_coll_name = partition_coll_value
[, partition_coll2_name = partition_coll2_value]
[,...])]
[LOCATION 'location2']
[,...]
Parameters

[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.

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 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.

Examples

ALTER TABLE customers PARTITION (zip='98040', state='WA') SET LOCATION 's3://mystorage/custdata';
ALTER TABLE SET TBLPROPERTIES

Adds custom metadata properties to a table sets their assigned values. Managed tables are not supported, so setting 'EXTERNAL'='FALSE' has no effect.

Synopsis

```
ALTER TABLE table_name SET TBLPROPERTIES ('property_name' = 'property_value' [ , ... ])
```

Parameters

```
SET TBLPROPERTIES ('property_name' = 'property_value' [ , ... ])
```

Specifies the metadata properties to add as `property_name` and the value for each as `property_value`. If `property_name` already exists, its value is reset to `property_value`.

Examples

```
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

```
CREATE (DATABASE|SCHEMA) [IF NOT EXISTS] database_name
[COMMENT 'database_comment']
[LOCATION 'S3_loc']
[WITH DBPROPERTIES ('property_name' = 'property_value') [, ...]]
```

Parameters

```
[IF NOT EXISTS]
```

Causes the error to be suppressed if a database named `database_name` already exists.

```
[COMMENT database_comment]
```

Establishes the metadata value for the built-in metadata property named `comment` and the value you provide for `database_comment`.

```
[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 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] [, ...])]
[COMMENT table_comment]
[PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]
[ROW FORMAT row_format]
[STORED AS file_format]
[WITH SERDEPROPERTIES (...)]
[LOCATION 's3_loc']
[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. 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 `my_column`.

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](#).

• **BINARY (for data in Parquet)**

• Date and time types

• **DATE, in the UNIX format, such as YYYY-MM-DD.**

• **TIMESTAMP.** Instant in time and date in the UNIX format, such as `yyyy-mm-dd hh:mm:ss[.f...]`. For example, `TIMESTAMP '2008-09-15 03:04:05.324'`. This format uses the session time zone.

• **ARRAY < data_type >**

• **MAP < primitive_type, data_type >**

• **STRUCT < col_name : data_type [COMMENT col_comment] [, ...] >**

    **[COMMENT table_comment]**

    Creates the comment table property and populates it with the table_comment you specify.

    **[PARTITIONED BY (col_name data_type [ COMMENT col_comment ], ... )]**

    Creates a partitioned table with one or more partition columns that have the col_name, data_type and col_comment specified. A table can have one or more partitions, which consist of a distinct column name and value combination. A separate data directory is created for each specified combination, which can improve query performance in some circumstances. Partitioned columns don't exist within the table data itself. If you use a value for col_name that is the same as a table column, you get an error. For more information, see [Partitioning Data (p. 74)](#).

    **Note**

    After you create a table with partitions, run a subsequent query that consists of the **MSCK REPAIR TABLE (p. 230)** clause to refresh partition metadata, for example, `MSCK REPAIR TABLE cloudfront_logs`.

    **[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] -- (Note: Available in Hive 0.13 and later)

--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_loc']

Specifies the location of the underlying data in Amazon S3 from which the table is created, for example, 's3://mystorage/'. For more information about considerations such as data format and permissions, see Requirements for Tables in Athena and Data in Amazon S3 (p. 68).

Use a trailing slash for your folder or bucket. Do not use file names or glob characters.

Use: s3://mybucket/key/

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 Configuring Encryption Options (p. 62).

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. 40) and Authoring Jobs in Glue in the AWS Glue Developer Guide.

Examples

```
CREATE EXTERNAL TABLE IF NOT EXISTS mydatabase.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+\([^\s\!?:]+\)$"
) LOCATION 's3://athena-examples/cloudfront/plaintext/';
```

CREATE TABLE AS

Creates a new table populated with the results of a SELECT (p. 234) query. To create an empty table, use CREATE TABLE (p. 222).

Topics

• Synopsis (p. 218)
• CTAS Table Properties (p. 226)

Synopsis

```
CREATE TABLE table_name
[ WITH ( property_name = expression [, ...] ) ]
AS query
[ WITH [ NO ] DATA ]
```

Where:

**WITH ( property_name = expression [, ...] )**

A list of optional CTAS table properties, some of which are specific to the data storage format. See CTAS Table Properties (p. 226).

query

A SELECT (p. 234) 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.
CREATE TABLE AS

[ 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. 97).

WITH (property_name = expression [, ...], )

external_location = [location]

The location where Athena saves your CTAS query in Amazon S3, for example, WITH (external_location = 's3://my-bucket/tables/parquet_table/'). This property is optional. When you don't specify any location and your workgroup does not override client-side settings (p. 166), Athena stores the CTAS query results in external_location = 's3://aws-athena-query-results-<account>-<region>/<query-name-or-unsaved>/<year>/<month>/<date>/<query-id>/', and does not use the same path again. If you specify the location manually, make sure that the Amazon S3 location has no data. Athena never attempts to delete your data. If you want to use the same location again, manually clean the data, otherwise your CTAS query will fail.

If the workgroup in which a query will run is configured with an enforced query results location (p. 166), 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 enforce 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;

To obtain the results location specified for the workgroup, view workgroup's details (p. 169).

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.

partitioned_by = ( [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( [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.
**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.

The optional `OR REPLACE` clause lets you update the existing view by replacing it. For more information, see Creating Views (p. 89).

**Synopsis**

```
CREATE [ OR REPLACE ] VIEW view_name AS query
```

**Examples**

To create a view `test` from the table `orders`, use a query similar to the following:

```
CREATE VIEW test AS
SELECT orderkey, orderstatus, totalprice / 2 AS half
FROM orders
```

To create a view `orders_by_date` from the table `orders`, use the following query:

```
CREATE VIEW orders_by_date AS
SELECT orderdate, sum(totalprice) AS price
FROM orders
GROUP BY orderdate
```

To update an existing view, use an example similar to the following:

```
CREATE OR REPLACE VIEW test AS
SELECT orderkey, orderstatus, totalprice / 4 AS quarter
FROM orders
```

See also SHOW COLUMNS (p. 230), SHOW CREATE VIEW (p. 231), DESCRIBE VIEW (p. 228), and DROP VIEW (p. 229).

**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]

Lists the metadata for the partition with partition_spec if included.

[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 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. 230), SHOW CREATE VIEW (p. 231), SHOW VIEWS (p. 233), and DROP VIEW (p. 229).

DROP DATABASE

Removes the named database from the catalog. If the database contains tables, you must either drop the tables before executing 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;

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. 91).

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. 227), SHOW COLUMNS (p. 230), SHOW CREATE VIEW (p. 231), SHOW VIEWS (p. 233), and DESCRIBE VIEW (p. 228).

**MSCK REPAIR TABLE**

Recovers partitions and data associated with partitions. Use this statement when you add partitions to the catalog. 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 the statement on the same table until all partitions are added. For more information, see Partitioning Data (p. 74).

**Synopsis**

```
MSCK REPAIR TABLE table_name
```

**Examples**

```
MSCK REPAIR TABLE orders;
```

**SHOW COLUMNS**

Lists the columns in the schema for a base table or a view.

**Synopsis**

```
SHOW COLUMNS IN table_name|view_name
```

**Examples**

```
SHOW COLUMNS IN clicks;
```

**SHOW CREATE TABLE**

Analyzes an existing table named `table_name` to generate the query that created it.

**Synopsis**

```
SHOW CREATE TABLE [db_name.]table_name
```

**Parameters**

```
TABLE [db_name.]table_name
```

The `db_name` parameter is optional. If omitted, the context defaults to the current database.

**Note**

The table name is required.
SHOW CREATE VIEW

Shows the SQL statement that creates the specified view.

Synopsis

SHOW CREATE VIEW view_name

Examples

SHOW CREATE VIEW orderclickstoday;

SHOW CREATE TABLE `salesdata.orderclickstoday`;

SHOW CREATE VIEW orders_by_date

See also CREATE VIEW (p. 227) and DROP VIEW (p. 229).

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 you specify. Wildcards can only be *, which indicates any character, or |, which indicates a choice between characters.

Examples

SHOW SCHEMAS;

SHOW DATABASES LIKE '*analytics';

SHOW PARTITIONS

Lists all the partitions in a table.
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

SHOW TABLES;

SHOW TABLES IN marketing_analytics 'orders*';

SHOW TBLPROPERTIES

Lists table properties for the named table.

Synopsis

SHOW TBLPROPERTIES table_name [('property_name')]

Parameters

[('property_name')]

If included, only the value of the property named property_name is listed.
SHOW VIEWS

Lists the views in the specified database, or in the current database if you omit the database name. Use the optional LIKE clause with a regular expression to restrict the list of view names.

Athena returns a list of STRING type values where each value is a view name.

Examples

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

SHOW VIEWS [IN database_name] LIKE ['regular_expression']

Parameters

[IN database_name]

Specifies the database_name from which views will be listed. If omitted, the database from the current context is assumed.

[LIKE 'regular_expression']

Filters the list of views to those that match the regular_expression you specify. Only the wildcard *, which indicates any character, or |, which indicates a choice between characters, can be used.

Examples

SHOW VIEWS;

SHOW VIEWS IN marketing-analytics LIKE 'orders*';

See also SHOW COLUMNS (p. 230), SHOW CREATE VIEW (p. 231), DESCRIBE VIEW (p. 228), and DROP VIEW (p. 229).

SQL Queries, Functions, and Operators

Use the following functions directly in Athena.

Amazon Athena query engine is based on Presto 0.172. For more information about these functions, see Presto 0.172 Functions and Operators.

Athena does not support all of Presto's features. For information, see Limitations (p. 238).

• SELECT (p. 234)
SELECT

Retrieves rows from zero or more tables.

Synopsis

[ WITH with_query [, ...] ]
SELECT [ ALL | DISTINCT ] select_expression [, ...]
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY [ ALL | DISTINCT ] grouping_element [, ...] ]
[ HAVING condition ]
[ UNION [ ALL | DISTINCT ] union_query ]
[ ORDER BY expression [ ASC | DESC ] [ NULLS FIRST | NULLS LAST] [, ... ] ]
[ LIMIT [ count | ALL ] ]

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:

```sql
subquery_table_name [ ( column_name [, ...] ) ] AS (subquery)
```

Where:
- `subquery_table_name` is a unique name for a temporary table that defines the results of the WITH clause subquery. Each subquery must have a table name that can be referenced in the FROM clause.
- `column_name [, ...]` is an optional list of output column names. The number of column names must be equal to or less than the number of columns defined by subquery.
- `subquery` is any query statement.

[ **ALL** | **DISTINCT** ] `select_expr`

`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.

[ **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.
The `grouping_expressions` element can be any function, such as `SUM`, `AVG`, or `COUNT`, performed on input columns, or be an ordinal number that selects an output column by position, starting at one.

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.

These complex grouping operations don't support expressions comprising input columns. Only column names or ordinals 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.

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.

[ 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 [ ALL | DISTINCT ] union_query ]

Combines the results of more than one SELECT statement into a single query. ALL or DISTINCT control which rows are included in the final result set.

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. DISTINCT is the default.

Multiple UNION clauses are processed left to right unless you use parentheses to explicitly define the order of processing.

[ 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.

Examples

```
SELECT * FROM table;
```

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

For more examples, see Querying Data in Amazon Athena Tables (p. 83).

Unsupported DDL

The following native Hive DDLs 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 ADD COLUMNS
- ALTER TABLE table_name partitionSpec CHANGE COLUMNS
Limitations

Athena does not support the following features, which are supported by an open source Presto version 0.172.
• User-defined functions (UDFs or UDAFs).
• Stored procedures.
• A particular subset of data types is supported. For more information, see Data Types (p. 216).
• INSERT INTO statements.
• The maximum number of partitions you can create in the CREATE TABLE AS SELECT (CTAS) statements is 100. For information, see CREATE TABLE AS (p. 225).
• PREPARED statements. You cannot run EXECUTE WITH USING.
• CREATE TABLE LIKE.
• DESCRIBE INPUT and DESCRIBE OUTPUT.
• EXPLAIN statements.
• Federated connectors. For more information, see Connectors.
• When you query columns with complex data types (array, map, struct), and are using Parquet for storing data, Athena currently reads an entire row of data, instead of selectively reading only the specified columns as expected. This is a known issue.
Code Samples, Service Limits, and Previous JDBC Driver

Use the following examples to create Athena applications based on AWS SDK for Java.

Use the links in this section to use the previous version of the JDBC driver.

Learn about service limits.

Topics
- Code Samples (p. 240)
- Using the Previous Version of the JDBC Driver (p. 248)
- Service Limits (p. 253)

Code Samples

Use examples in this topic as a starting point for writing Athena applications using the AWS SDK for Java.

- Java Code Samples
  - Create a Client to Access Athena (p. 240)
  - Working with Query Executions
    - Start Query Execution (p. 241)
    - Stop Query Execution (p. 244)
    - List Query Executions (p. 245)
  - Working with Named Queries
    - Create a Named Query (p. 246)
    - Delete a Named Query (p. 246)
    - List Query Executions (p. 245)

Note
These samples use constants (for example, ATHENA_SAMPLE_QUERY) for strings, which are defined in an ExampleConstants class declaration not shown in this topic. Replace these constants with your own strings or defined constants.

Create a Client to Access Athena

```java
package com.amazonaws.services.athena.sdksamples;

import com.amazonaws.ClientConfiguration;
import com.amazonaws.auth.InstanceProfileCredentialsProvider;
import com.amazonaws.regionsRegions;
import com.amazonaws.services.athena.AmazonAthena;
import com.amazonaws.services.athena.AmazonAthenaClientBuilder;

/**
* AthenaClientFactory
* ---------------------
*/
Star Query Execution

```java
package com.amazonaws.services.athena.sdksamples;

import com.amazonaws.services.athena.AmazonAthena;
import com.amazonaws.services.athena.model.ColumnInfo;
import com.amazonaws.services.athena.model.GetQueryExecutionRequest;
import com.amazonaws.services.athena.model.GetQueryExecutionResult;
import com.amazonaws.services.athena.model.GetQueryResultsRequest;
import com.amazonaws.services.athena.model.GetQueryResultsResult;
import com.amazonaws.services.athena.model.QueryExecutionContext;
import com.amazonaws.services.athena.model.QueryExecutionState;
import com.amazonaws.services.athena.model.ResultConfiguration;
import com.amazonaws.services.athena.model.Row;
import com.amazonaws.services.athena.model.StartQueryExecutionRequest;
import com.amazonaws.services.athena.model.StartQueryExecutionResult;
import java.util.List;

/**
 * StartQueryExample
 * -------------------------------------
 * This code shows how to submit a query to Athena for execution, wait till results
 * are available, and then process the results.
 * */
public class StartQueryExample {
    public static void main(String[] args) throws InterruptedException {
        // Build an AmazonAthena client
        AthenaClientFactory factory = new AthenaClientFactory();
        AmazonAthena client = factory.createClient();

        String queryExecutionId = submitAthenaQuery(client);

        waitForQueryToComplete(client, queryExecutionId);

        processResultRows(client, queryExecutionId);
    }
}
```
* Submits a sample query to Athena and returns the execution ID of the query.

```java
private static String submitAthenaQuery(AmazonAthena client)
{
    // The QueryExecutionContext allows us to set the Database.
    QueryExecutionContext queryExecutionContext = new QueryExecutionContext().withDatabase(ExampleConstants.ATHENA_DEFAULT_DATABASE);

    // The result configuration specifies where the results of the query should go in S3
    // and encryption options
    ResultConfiguration resultConfiguration = new ResultConfiguration()
        .withEncryptionConfiguration(encryptionConfiguration)
        .withOutputLocation(ExampleConstants.ATHENA_OUTPUT_BUCKET);

    // Create the StartQueryExecutionRequest to send to Athena which will start the
    // query.
    StartQueryExecutionRequest startQueryExecutionRequest = new StartQueryExecutionRequest()
            .withQueryString(ExampleConstants.ATHENA_SAMPLE_QUERY)
            .withQueryExecutionContext(queryExecutionContext)
            .withResultConfiguration(resultConfiguration);

    StartQueryExecutionResult startQueryExecutionResult = client.startQueryExecution(startQueryExecutionRequest);
    return startQueryExecutionResult.getQueryExecutionId();
}
```

/**
 * Wait for an Athena query to complete, fail or to be cancelled. This is done by polling
 * Athena over an
 * interval of time. If a query fails or is cancelled, then it will throw an exception.
 *
 */

```java
private static void waitForQueryToComplete(AmazonAthena client, String queryExecutionId) throws InterruptedException
{
    GetQueryExecutionRequest getQueryExecutionRequest = new GetQueryExecutionRequest()
            .withQueryExecutionId(queryExecutionId);

    GetQueryExecutionResult getQueryExecutionResult = null;
    boolean isQueryStillRunning = true;
    while (isQueryStillRunning) {
        getQueryExecutionResult = client.getQueryExecution(getQueryExecutionRequest);
        String queryState = getQueryExecutionResult.getQueryExecution().getStatus().getState();
        if (queryState.equals(QueryExecutionState.FAILED.toString())) {
            throw new RuntimeException("Query Failed to run with Error Message: " + getQueryExecutionResult.getQueryExecution().getStatus().getStateChangeReason());
        } else if (queryState.equals(QueryExecutionState.CANCELLED.toString())) {
            throw new RuntimeException("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("Current Status is: " + queryState);
    }
}
```
private static void processResultRows(AmazonAthena client, String queryExecutionId) {
    GetQueryResultsRequest getQueryResultsRequest = new GetQueryResultsRequest()
        .withMaxResults(1000)
        .withQueryExecutionId(queryExecutionId);

    GetQueryResultsResult getQueryResultsResult = client.getQueryResults(getQueryResultsRequest);
    List<ColumnInfo> columnInfoList = getQueryResultsResult ResultSet().getResultSetMetadata().getColumnInfo();

    while (true) {
        List<Row> results = getQueryResultsResult.ResultSet().getRows();
        for (Row row : results) {
            // Process the row. The first row of the first page holds the column names.
            processRow(row, columnInfoList);
        }
        // If nextToken is null, there are no more pages to read. Break out of the loop.
        if (getQueryResultsResult.getNextToken() == null) {
            break;
        }
        getQueryResultsResult = client.getQueryResults(
            getQueryResultsRequest.withNextToken(getQueryResultsResult getNextToken()));
    }
}

private static void processRow(Row row, List<ColumnInfo> columnInfoList) {
    for (int i = 0; i < columnInfoList.size(); ++i) {
        switch (columnInfoList.get(i).getType()) {
            case "varchar":
                // Convert and Process as String
                break;
            case "tinyint":
                // Convert and Process as tinyint
                break;
            case "smallint":
                // Convert and Process as smallint
                break;
            case "integer":
                // Convert and Process as integer
                break;
            case "bigint":
                // Convert and Process as bigint
                break;
            case "double":
                // Convert and Process as double
                break;
            case "boolean":
                // Convert and Process as boolean
                break;
            case "date":
                // Convert and Process as date
                break;
            case "timestamp":
                // Convert and Process as timestamp
                break;
            default:
                // Convert and Process as columnInfoList.get(i).getType()
                break;
        }
    }
}
throw new RuntimeException("Unexpected Type is not expected" +
columnInfoList.get(i).getType());
    }
}  
}
}

Stop Query Execution

```java
package com.amazonaws.services.athena.sdksamples;

import com.amazonaws.services.athena.AmazonAthena;
import com.amazonaws.services.athena.model.GetQueryExecutionRequest;
import com.amazonaws.services.athena.model.GetQueryExecutionResult;
import com.amazonaws.services.athena.model.QueryExecutionContext;
import com.amazonaws.services.athena.model.ResultConfiguration;
import com.amazonaws.services.athena.model.StartQueryExecutionRequest;
import com.amazonaws.services.athena.model.StartQueryExecutionResult;
import com.amazonaws.services.athena.model.StopQueryExecutionRequest;
import com.amazonaws.services.athena.model.StopQueryExecutionResult;

/**
 * StopQueryExecutionExample
 * -------------------------------------
 * This code runs an example query, immediately stops the query, and checks the status of
 * the query to
 * ensure that it was cancelled.
 */
public class StopQueryExecutionExample {
    public static void main(String[] args) throws Exception {
        // Build an Athena client
        AthenaClientFactory factory = new AthenaClientFactory();
        AmazonAthena client = factory.createClient();

        String sampleQueryExecutionId = getExecutionId(client);

        // Submit the stop query Request
        StopQueryExecutionRequest stopQueryExecutionRequest = new StopQueryExecutionRequest()
                .withQueryExecutionId(sampleQueryExecutionId);

        StopQueryExecutionResult stopQueryExecutionResult =
                client.stopQueryExecution(stopQueryExecutionRequest);

        // Ensure that the query was stopped
        GetQueryExecutionRequest getQueryExecutionRequest = new GetQueryExecutionRequest()
                .withQueryExecutionId(sampleQueryExecutionId);

        GetQueryExecutionResult getQueryExecutionResult =
                client.getQueryExecution(getQueryExecutionRequest);

        if (getQueryExecutionResult.getQueryExecution().getStatus().getState().equals(ExampleConstants.QUERY_STATE_CANCELLED)) {
            // Query was cancelled.
            System.out.println("Query has been cancelled");
        }
    }
}

/**
 * Submits an example query and returns a query execution ID of a running query to stop.
 */
public static String getExecutionId(AmazonAthena client) {
    // Build an Athena client
    AthenaClientFactory factory = new AthenaClientFactory();
    AmazonAthena client = factory.createClient();

    String sampleQueryExecutionId = getExecutionId(client);

    // Submit the stop query Request
    StopQueryExecutionRequest stopQueryExecutionRequest = new StopQueryExecutionRequest()
            .withQueryExecutionId(sampleQueryExecutionId);

    StopQueryExecutionResult stopQueryExecutionResult =
            client.stopQueryExecution(stopQueryExecutionRequest);

    // Ensure that the query was stopped
    GetQueryExecutionRequest getQueryExecutionRequest = new GetQueryExecutionRequest()
            .withQueryExecutionId(sampleQueryExecutionId);

    GetQueryExecutionResult getQueryExecutionResult =
            client.getQueryExecution(getQueryExecutionRequest);

    if (getQueryExecutionResult.getQueryExecution().getStatus().getState().equals(ExampleConstants.QUERY_STATE_CANCELLED)) {
        // Query was cancelled.
        System.out.println("Query has been cancelled");
    }
} 
```
List Query Executions

```java
package com.amazonaws.services.athena.sdksamples;

import com.amazonaws.services.athena.AmazonAthena;
import com.amazonaws.services.athena.model.ListQueryExecutionsRequest;
import com.amazonaws.services.athena.model.ListQueryExecutionsResult;
import java.util.List;

/**
 * ListQueryExecutionsExample
 * -------------------------------------
 * This code shows how to obtain a list of query execution IDs.
 */
public class ListQueryExecutionsExample {

    public static void main(String[] args) throws Exception {
        // Build an Athena client
        AthenaClientFactory factory = new AthenaClientFactory();
        AmazonAthena client = factory.createClient();

        // Build the request
        ListQueryExecutionsRequest listQueryExecutionsRequest = new ListQueryExecutionsRequest();

        // Get the list results.
        ListQueryExecutionsResult listQueryExecutionsResult = client.listQueryExecutions(listQueryExecutionsRequest);

        // Process the results.
        boolean hasMoreResults = true;
        while (hasMoreResults) {
            List<String> queryExecutionIds = listQueryExecutionsResult.getQueryExecutionIds();
            // process queryExecutionIds.

            // If nextToken is not null, then there are more results. Get the next page of results.
            if (listQueryExecutionsResult.getNextToken() != null) {
                listQueryExecutionsResult = client.listQueryExecutions(
```
Create a Named Query

```java
package com.amazonaws.services.athena.sdksamples;

import com.amazonaws.services.athena.AmazonAthena;
import com.amazonaws.services.athena.model.CreateNamedQueryRequest;
import com.amazonaws.services.athena.model.CreateNamedQueryResult;

/**
 * CreateNamedQueryExample
 * -------------------------------------
 * This code shows how to create a named query.
 */
public class CreateNamedQueryExample {
    public static void main(String[] args) throws Exception {
        // Build an Athena client
        AthenaClientFactory factory = new AthenaClientFactory();
        AmazonAthena client = factory.createClient();

        // Create the named query request.
        CreateNamedQueryRequest createNamedQueryRequest = new CreateNamedQueryRequest()
            .withDatabase(ExampleConstants.ATHENA_DEFAULT_DATABASE)
            .withQueryString(ExampleConstants.ATHENA_SAMPLE_QUERY)
            .withDescription("Sample Description")
            .withName("SampleQuery");

        // Call Athena to create the named query. If it fails, an exception is thrown.
        CreateNamedQueryResult createNamedQueryResult = client.createNamedQuery(createNamedQueryRequest);
    }
}
```

Delete a Named Query

```java
package com.amazonaws.services.athena.sdksamples;

import com.amazonaws.services.athena.AmazonAthena;
import com.amazonaws.services.athena.model.CreateNamedQueryRequest;
import com.amazonaws.services.athena.model.CreateNamedQueryResult;
import com.amazonaws.services.athena.model.DeleteNamedQueryRequest;
import com.amazonaws.services.athena.model.DeleteNamedQueryResult;

/**
 * DeleteNamedQueryExample
 * -------------------------------------
 * This code shows how to delete a named query by using the named query ID.
 */
public class DeleteNamedQueryExample {
}
```
private static String getNamedQueryId(AmazonAthena athenaClient) {
    // Create the NameQuery Request.
    CreateNamedQueryRequest createNamedQueryRequest = new CreateNamedQueryRequest()
        .withDatabase(ExampleConstants.ATHENA_DEFAULT_DATABASE)
        .withQueryString(ExampleConstants.ATHENA_SAMPLE_QUERY)
        .withName("SampleQueryName")
        .withDescription("Sample Description");

    // Create the named query. If it fails, an exception is thrown.
    CreateNamedQueryResult createNamedQueryResult = athenaClient.createNamedQuery(createNamedQueryRequest);
    return createNamedQueryResult.getNamedQueryId();
}

public static void main(String[] args) throws Exception {
    // Build an Athena client
    AthenaClientFactory factory = new AthenaClientFactory();
    AmazonAthena client = factory.createClient();

    String sampleNamedQueryId = getNamedQueryId(client);

    // Create the delete named query request
    DeleteNamedQueryRequest deleteNamedQueryRequest = new DeleteNamedQueryRequest()
        .withNamedQueryId(sampleNamedQueryId);

    // Delete the named query
    DeleteNamedQueryResult deleteNamedQueryResult = client.deleteNamedQuery(deleteNamedQueryRequest);
}

List Named Queries

package com.amazonaws.services.athena.sdksamples;
import com.amazonaws.services.athena.AmazonAthena;
import com.amazonaws.services.athena.model.ListNamedQueriesRequest;
import com.amazonaws.services.athena.model.ListNamedQueriesResult;
import java.util.List;

/**
   * ListNamedQueryExample
   * -------------------------------------
   * This code shows how to obtain a list of named query IDs.
   */
public class ListNamedQueryExample {
    public static void main(String[] args) throws Exception {
        // Build an Athena client
        AthenaClientFactory factory = new AthenaClientFactory();
        AmazonAthena client = factory.createClient();

        // Build the request
        ListNamedQueriesRequest listNamedQueriesRequest = new ListNamedQueriesRequest();

        // Get the list results.
        ListNamedQueriesResult listNamedQueriesResult = client.listNamedQueries(listNamedQueriesRequest);
Using the Previous Version of the JDBC Driver

We recommend that you use the current version of the JDBC driver, which is version 2.0.7. For information, see Using Athena with the JDBC Driver (p. 43). If you need to use the previous versions, follow the steps in this section to download and configure the driver.

The previous versions of the JDBC driver are 2.0.6, 2.0.5, and 2.0.2.

The JDBC driver version 1.1.0 is also available for download, however, 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.

Using the Previous Version of the JDBC Driver

1. Download the version of the driver that you need.

Links for Downloading Previous Versions of the JDBC Driver

<table>
<thead>
<tr>
<th>Driver Version 2.0.6</th>
<th>Download Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC 2.0.6 compatible with JDBC 4.1 and JDK 7.0 or later</td>
<td>AthenaJDBC41-2.0.6.jar</td>
</tr>
<tr>
<td>JDBC 2.0.6 compatible with JDBC 4.2 and JDK 8.0 or later</td>
<td>AthenaJDBC42-2.0.6.jar</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Driver Version 2.0.5</th>
<th>Download Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC 2.0.5 compatible with JDBC 4.1 and JDK 7.0 or later</td>
<td>AthenaJDBC41-2.0.5.jar</td>
</tr>
<tr>
<td>JDBC 2.0.5 compatible with JDBC 4.2 and JDK 8.0 or later</td>
<td>AthenaJDBC42-2.0.5.jar</td>
</tr>
</tbody>
</table>
Instructions for JDBC Driver version 1.1.0

This section includes instructions for the 1.1.0 version of the JDBC driver. Use these instructions only if you have not migrated to the newer (and supported) version of the JDBC driver.

Download the JDBC driver version 1.1.0 that is compatible with JDBC 4.1 and JDK 7.0: 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. 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`
- `athena:GetTable`
- `athena:GetTables`

**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>, which is established under Settings in the Athena Console, <a href="https://console.aws.amazon.com/athena/">https://console.aws.amazon.com/athena/</a>. The JDBC driver then asks Athena to read the results and provide rows of data back to the user.</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 s3_staging_dir. 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 query_results_encryption_option specifies SSE-KMS or CSE-KMS. For example, 123abcde-4e56-56f7-g890-1234h5678i9j.</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>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>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:

```java
public CustomSessionsCredentialsProvider(String accessId, String secretKey, String token) {
    //...
}
```

You would then set the properties as follows:

```java
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>
</tbody>
</table>
### Service Limits

**Note**

You can contact AWS Support to request a limit increase for the limits listed here.

- By default, limits on your account allow you to submit:
  - 20 DDL queries at the same time. DDL queries include `CREATE TABLE` and `CREATE TABLE ADD PARTITION` queries.
  - 20 DML queries at the same time. DML queries include `SELECT` and `CREATE TABLE AS (CTAS)` queries.

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.

Athena service limits are shared across all workgroups in the account.

These are soft limits and you can request a limit increase. These limits in Athena are defined as the number of queries that can be submitted to the service at the same time. You can submit up to 20 queries of the same type (DDL or DML) at a time. If you submit a query that exceeds the query limit, the Athena API displays an error message: "You have exceeded the limit for the number of queries you can run concurrently. Reduce the number of concurrent queries submitted by this account. Contact customer support to request a concurrent query limit increase."

- If you use Athena in regions where AWS Glue is available, migrate to AWS Glue Data Catalog. See Upgrading to the AWS Glue Data Catalog Step-by-Step (p. 29). If you have migrated to AWS Glue, for service limits on tables, databases, and partitions in Athena, see AWS Glue Limits. If you have not migrated to AWS Glue Catalog, the number of partitions per table is 20,000. You can request a limit increase.

- You may encounter a limit for Amazon S3 buckets per account, which is 100. Athena also needs a separate bucket to log results.

- The query timeout is 30 minutes.

- The maximum allowed query string length is 262144 bytes, where the strings are encoded in UTF-8. Use these tips (p. 71) for naming columns, tables, and databases in Athena.

- Athena APIs have the following default limits 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>
</tbody>
</table>
Service Limits

<table>
<thead>
<tr>
<th>API Name</th>
<th>Default Number of Calls per Second</th>
<th>Burst Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetQueryExecution, GetQueryResults</td>
<td>100</td>
<td>up to 200</td>
</tr>
</tbody>
</table>

For example, for `StartQueryExecution`, you can make up to 20 calls per second. 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 limit 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. You can contact AWS Support to request a limit increase.
# Document History

This documentation is associated with the May, 18, 2017 version of Amazon Athena.

**Latest documentation update: January 24, 2019.**

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Release Date</th>
</tr>
</thead>
<tbody>
<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 “Querying Network Load Balancer Logs” (p. 142).</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 Athena API” (p. 59).</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 Access Control Policies.</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. 91), Considerations and Limitations (p. 91), and Examples (p. 97).</td>
<td>October 10, 2018</td>
</tr>
<tr>
<td>Released the ODBC driver version 1.0.3 with support for streaming results instead of fetching them in pages.</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 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. 44).</td>
<td>September 6, 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>Made other improvements in the documentation.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Released the JDBC driver version 2.0.5 with default support for streaming results instead of fetching them in pages.</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. 43).</td>
<td>August 16, 2018</td>
</tr>
<tr>
<td></td>
<td>For information about streaming results, search for UseResultsetStreaming in the JDBC Driver Installation and Configuration Guide.</td>
<td></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.</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. 145).</td>
<td>August 7, 2018</td>
</tr>
<tr>
<td></td>
<td>Updated examples for querying ALB logs. For information, see Querying Application Load Balancer Logs (p. 143).</td>
<td></td>
</tr>
<tr>
<td>Added support for views. Added guidelines for schema manipulations for various data storage formats.</td>
<td>Added support for views. For information, see Views (p. 86). Updated this guide with guidance on handling schema updates for various data storage formats. For information, see Handling Schema Updates (p. 148).</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 Limits (p. 253).</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. 27).</td>
<td>May 8, 2018</td>
</tr>
<tr>
<td>Released the JDBC driver version 2.0.2.</td>
<td>Released the new version of the JDBC driver (version 2.0.2). For information, see Using Athena with the JDBC Driver (p. 43).</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>Change</td>
<td>Description</td>
<td>Release Date</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Added an ability to create Athena tables</td>
<td>Added an ability to automatically create Athena tables for CloudTrail log files directly from the CloudTrail console. For information, see Creating a Table for CloudTrail Logs in the CloudTrail Console (p. 136).</td>
<td>March 15, 2018</td>
</tr>
<tr>
<td>for CloudTrail log files directly from the</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CloudTrail console.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Added support for securely offloading</td>
<td>Added an ability to securely offload intermediate data to disk for queries with the GROUP BY clause. This improves the reliability of such queries, preventing &quot;Query resource exhausted&quot; errors. For more information, see the release note for February 2, 2018 (p. 14).</td>
<td>February 2, 2018</td>
</tr>
<tr>
<td>intermediate data to disk for queries with</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GROUP BY.</td>
<td></td>
<td></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. 14).</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),</td>
<td>Added support for querying geospatial data, and for Asia Pacific (Seoul), Asia Pacific (Mumbai), EU (London) regions. For information, see Querying Geospatial Data and AWS Regions and Endpoints.</td>
<td>November 1, 2017</td>
</tr>
<tr>
<td>Asia Pacific (Mumbai), and EU (London)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>regions.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Added support for querying geospatial data.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Added support for EU (Frankfurt).</td>
<td>Added support for EU (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</td>
<td>Added support for creating named Athena queries with AWS CloudFormation. For more information, see AWS::Athena::NamedQuery in the AWS CloudFormation User Guide.</td>
<td>October 3, 2017</td>
</tr>
<tr>
<td>AWS CloudFormation.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Added support for Asia Pacific (Sydney).</td>
<td>Added support for Asia Pacific (Sydney). For a list of supported regions, see AWS Regions and Endpoints.</td>
<td>September 25, 2017</td>
</tr>
<tr>
<td>Added a section to this guide for querying</td>
<td>Added examples for Querying AWS Service Logs (p. 135) and for querying different types of data in Athena. For information, see Querying Data in Amazon Athena Tables (p. 83).</td>
<td>September 5, 2017</td>
</tr>
<tr>
<td>AWS Service logs and different types of</td>
<td></td>
<td></td>
</tr>
<tr>
<td>data, including maps, arrays, nested data,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>and data containing JSON.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Release Date</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Added support for 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 (CTRL + ↑/↓ using Windows, CMD + ↑/↓ using Mac).</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 EU (Ireland).</td>
<td>Added support for EU (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 Configuring Encryption Options (p. 62).</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. 195). • 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.</td>
<td>March 24, 2017</td>
</tr>
<tr>
<td>Added support for US East (Ohio).</td>
<td>Added support for Avro SerDe (p. 192) and OpenCSVSerDe for Processing CSV (p. 197), US East (Ohio), and bulk editing columns in the console wizard. Improved performance on large Parquet tables.</td>
<td>February 20, 2017</td>
</tr>
</tbody>
</table>

The initial release of the Amazon Athena User Guide. | November, 2016 |
For the latest AWS terminology, see the AWS Glossary in the AWS General Reference.