Amazon Athena

User Guide

Table of Contents

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.

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 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. 18) and What is AWS Glue in the *AWS Glue Developer Guide*.

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. 33).

You can 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 CreateNamedQuery in the *Amazon Athena API Reference*, and AWS::Athena::NamedQuery in the *AWS CloudFormation User Guide*.

Accessing Athena

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

- To get started with the console, see Getting Started (p. 14).
- To learn how to use JDBC, see Connecting to Amazon Athena with JDBC (p. 33).
- To use the Athena API, see the Amazon Athena API Reference.
- To use the CLI, install the AWS CLI and then type `aws athena help` from the command line to see available commands. For information about available commands, see the AWS Athena command line 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. 18). When AWS Glue creates a table, it registers it in its own AWS Glue Data Catalog. Athena uses the AWS Glue Data Catalog to store and retrieve this metadata, using it when you run queries to analyze the underlying dataset.

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. 19).

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 write queries in the Athena Query Editor, see Getting Started (p. 14).

- 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.

Release Notes

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

Contents

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.
- 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. 144) in this guide, and Presto 0.172 Functions.

Athena does not support all of Presto's features. For more information, see Limitations (p. 149).

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. 35).

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. 83) and AWS Regions and Endpoints.

October 19, 2017

Published on *2017-10-19*

Added support for EU (Frankfurt). For a list of supported regions, see AWS Regions and Endpoints.

October 3, 2017

Published on *2017-10-03*

Create named Athena queries with CloudFormation. For more information, see
AWS::Athena::NamedQuery in the *AWS CloudFormation User Guide*.

September 25, 2017

Published on *2017-09-25*

Added support for Asia Pacific (Sydney). For a list of supported regions, see AWS Regions and Endpoints.

August 14, 2017

Published on *2017-08-14*

Added integration with the AWS Glue Data Catalog and a migration wizard for updating from the Athena
managed data catalog to the AWS Glue Data Catalog. For more information, see Integration with AWS
Glue (p. 18).

August 4, 2017

Published on *2017-08-04*

Added support for Grok SerDe, which provides easier pattern matching for records in unstructured text
files such as logs. For more information, see Grok SerDe (p. 115). 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 AWS CLI Reference for Athena.
- 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. 33).
- 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:

Deprecated Policy-Specific Action	Corresponding Athena API Action
`athena:RunQuery`	`athena:StartQueryExecution`
`athena:CancelQueryExecution`	`athena:StopQueryExecution`
`athena:GetQueryExecutions`	`athena:ListQueryExecutions`

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. 41).

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 JDBC Driver (p. 33).
- 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. 112). 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. 109)
  - OpenCSVSerDe for Processing CSV (p. 114)

- **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**.

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

1. Sign in to the IAM console at https://console.aws.amazon.com/iam/.
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**.

Amazon Athena User Guide
To create an IAM user for yourself, add the user to the
administrators group, and create a password for the user

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):

```
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. 37).

> **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. 40). If the dataset or Athena query results are encrypted, you may need additional permissions. For more information, see Configuring Encryption Options (p. 41).

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 .

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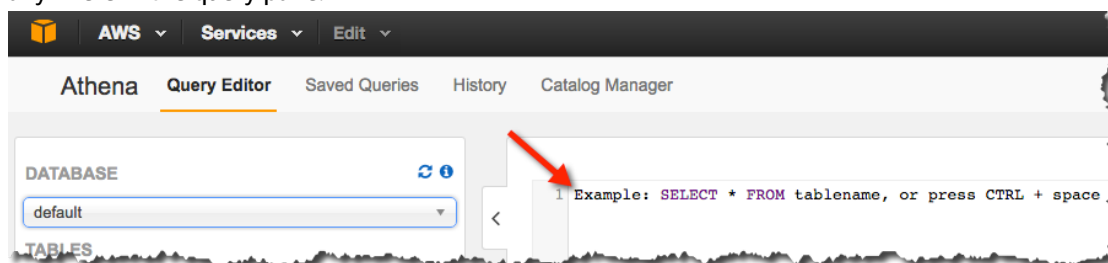
4. To create a database named `mydatabase`, enter the following CREATE DATABASE statement, and then choose **Run Query**:

```
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+[^\(]+[\(]([^\;]+).*\%20([^
\/]+)[\/]+)[\/](.*)$"
  ) 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**:

```
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:

	os	count
1	iOS	794
2	MacOS	852
3	OSX	799
4	Windows	883
5	Linux	813
6	Android	855

2. Optionally, you can save the results of a query to CSV by choosing the file icon on the **Results** pane.
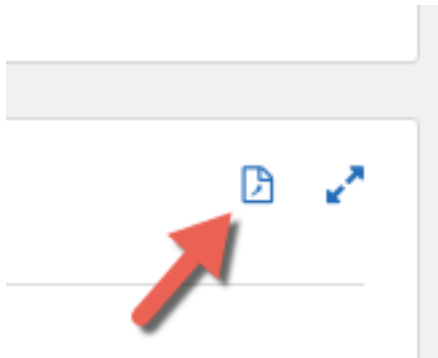
You can also view the results of previous queries or queries that may take some time to complete. Choose **History** then either search for your query or choose **View** or **Download** to view or download the results of previous completed queries. This also displays the status of queries that are currently running. Query history is retained for 45 days. For information, see Viewing Query History (p. 60).
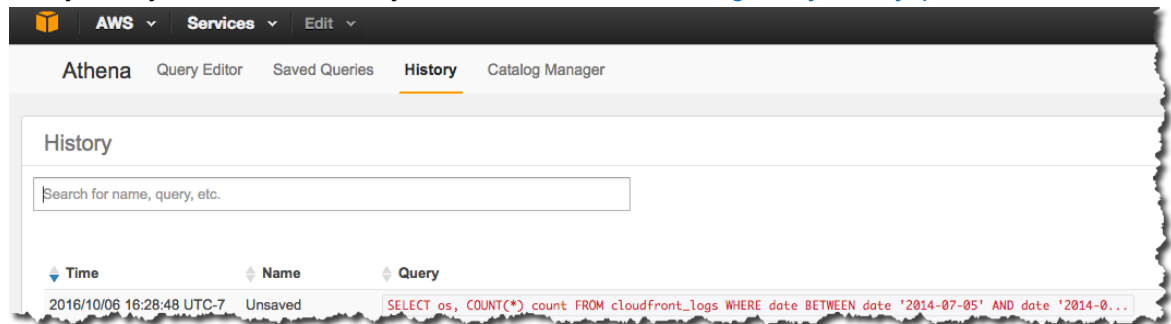
Query results are also stored in Amazon S3 in a bucket called aws-athena-query-results-*ACCOUNTID-REGION*. You can change the default location in the console and encryption options by choosing **Settings** in the upper right pane. For more information, see Query Results (p. 59).

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.
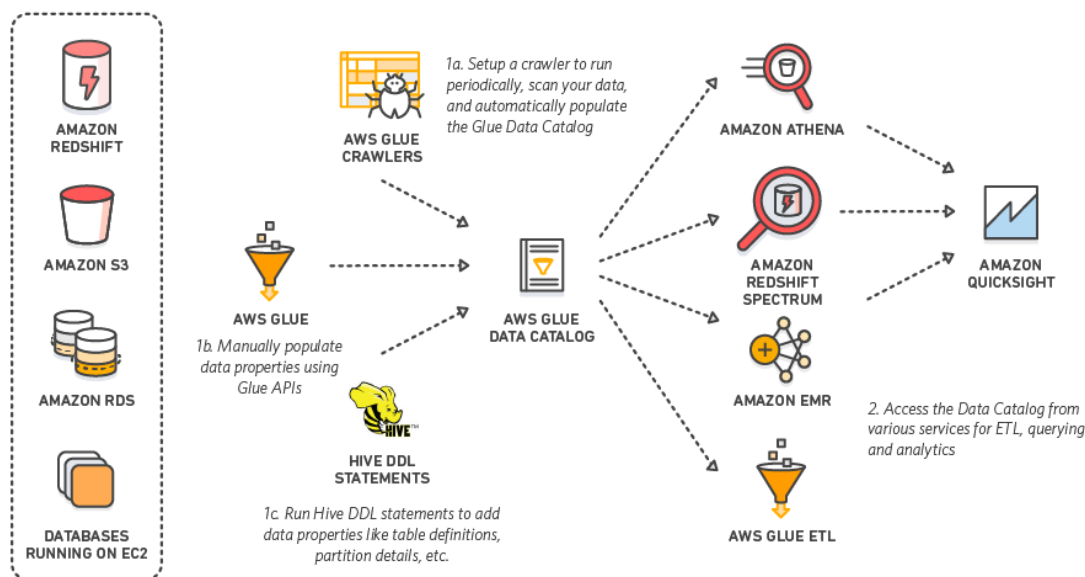
In regions where AWS Glue is supported, Athena uses the AWS Glue Data Catalog as a central location to store and retrieve table metadata throughout an AWS account. The Athena execution engine requires table metadata that instructs it where to read data, how to read it, and other information necessary 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. 22) For more information about the benefits of using AWS Glue with Athena, see Why should I upgrade to the AWS Glue Data Catalog? (p. 21)

Topics

- Upgrading to the AWS Glue Data Catalog Step-by-Step (p. 19)
- FAQ: Upgrading to the AWS Glue Data Catalog (p. 21)
- Best Practices When Using Athena with AWS Glue (p. 23)

Upgrading to the AWS Glue Data Catalog Step-by-Step

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. 21). 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. 23).

Step 1 - Allow a User to Perform the Upgrade

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

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

The following is an example policy statement.

```
{
    "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. 37).

```
{
  "Effect":"Allow",
  "Action":[
    "glue:CreateDatabase",
    "glue:DeleteDatabase",
    "glue:GetDatabase",
    "glue:GetDatabases",
    "glue:UpdateDatabase",
    "glue:CreateTable",
    "glue:DeleteTable",
    "glue:BatchDeleteTable",
    "glue:UpdateTable",
    "glue:GetTable",
    "glue:GetTables",
    "glue:BatchCreatePartition",
    "glue:CreatePartition",
    "glue:DeletePartition",
    "glue:BatchDeletePartition",
    "glue:UpdatePartition",
    "glue:GetPartition",
    "glue:GetPartitions",
    "glue:BatchGetPartition"
  ],
  "Resource":[
    "*"
  ]
}
```

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.
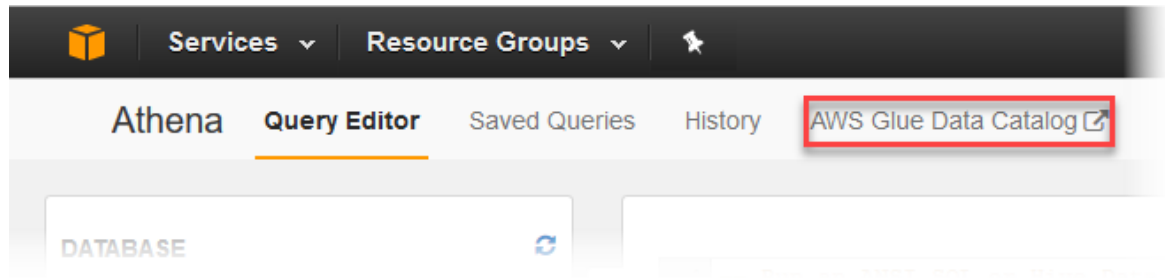
When you create a table using the console, you now have the option to create a table using an AWS Glue crawler. For more information, see Using AWS Glue Crawlers (p. 24).
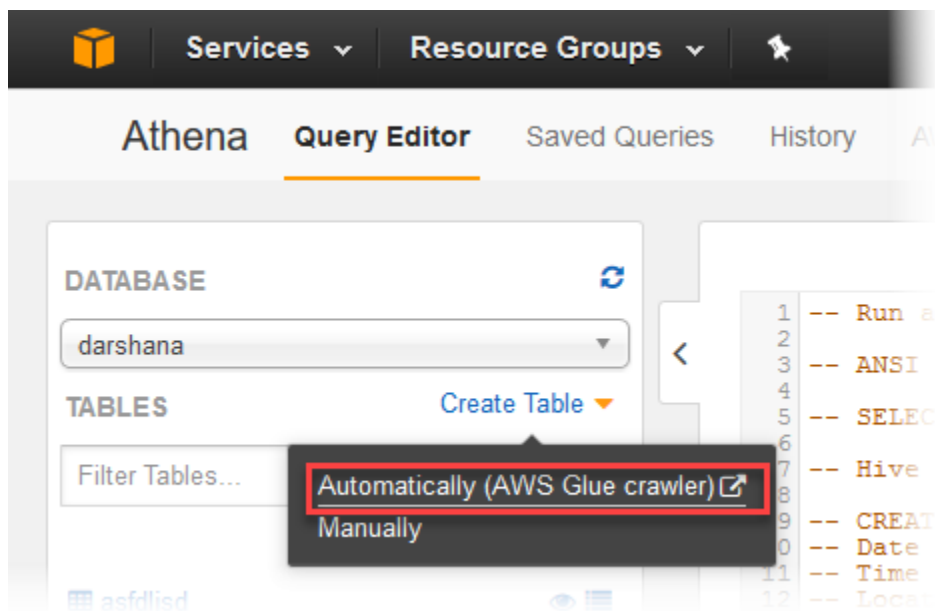
FAQ: Upgrading to the AWS Glue Data Catalog

If you created databases and tables using Athena in a region before AWS Glue was available in that region, metadata is stored in an Athena-managed data catalog, which only Athena and Amazon Redshift Spectrum can access. To use AWS Glue 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. 22)
- My users use a managed policy with Athena and Redshift Spectrum. What steps do I need to take to upgrade? (p. 22)
- What happens if I don't upgrade? (p. 23)
- Why do I need to add AWS Glue policies to Athena users? (p. 23)
- What happens if I don't allow AWS Glue policies for Athena users? (p. 23)
- Is there risk of data loss during the upgrade? (p. 23)
- Is my data also moved during this upgrade? (p. 23)

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. 19).

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. 19).

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

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*.
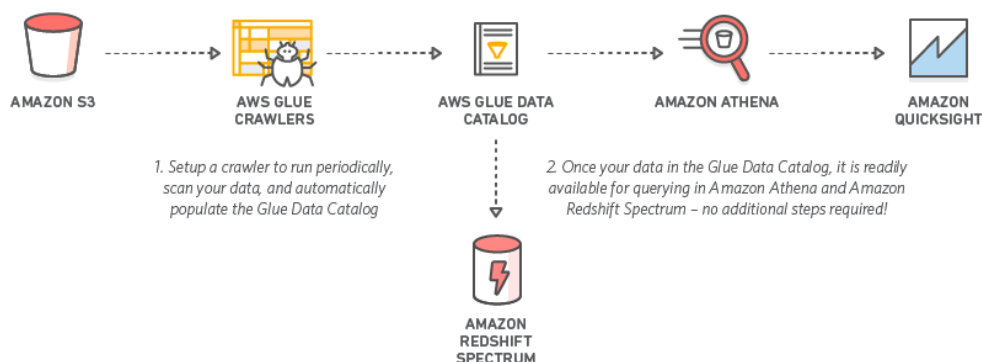
Using Multiple Data Sources with Crawlers

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 as intended, use the AWS Glue console to set the crawler to have two data sources, `s3://bucket01/folder1/table1/` and `s3://bucket01/folder1/table2`, as shown in the following procedure.

To add another data store to an existing crawler in AWS Glue

1. In the AWS Glue console, choose **Crawlers**, select your crawler, and then choose **Action**, **Edit crawler**.
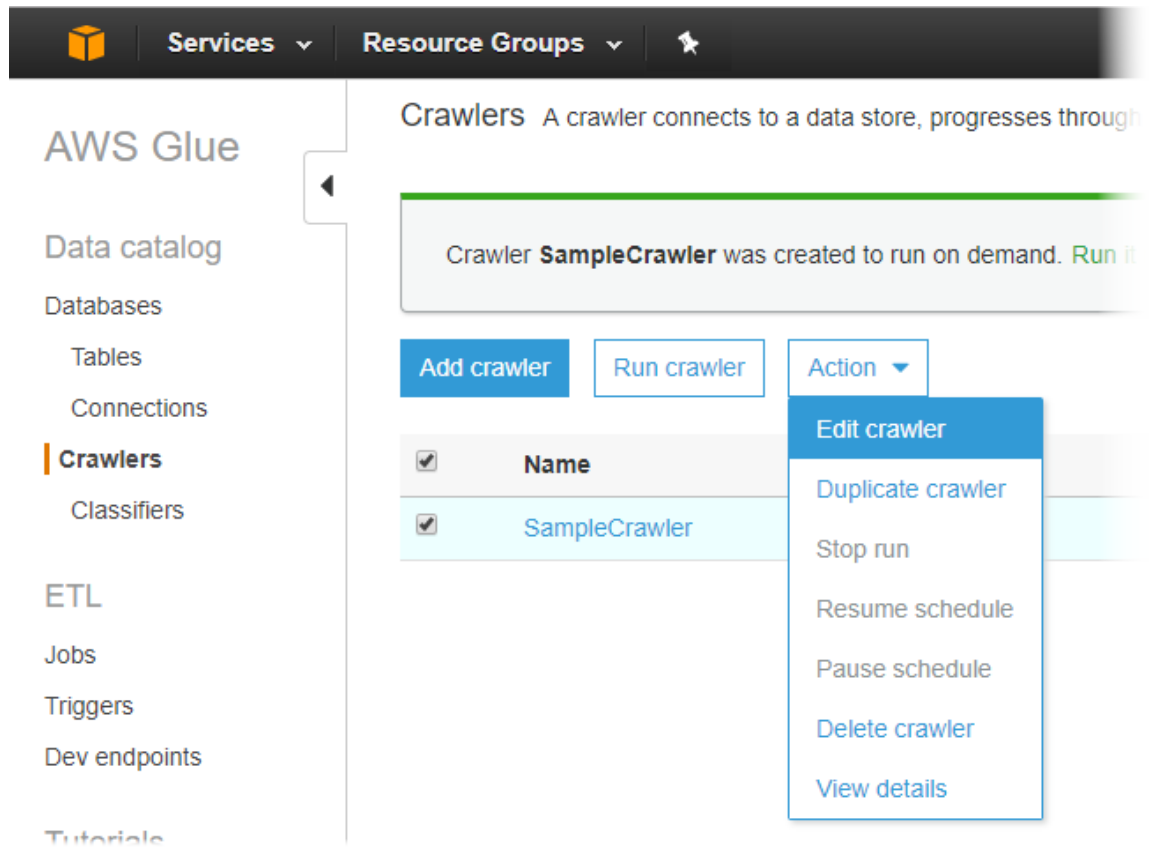
2. Under **Add information about your crawler**, choose additional settings as appropriate, and then choose **Next**.

3. 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**.
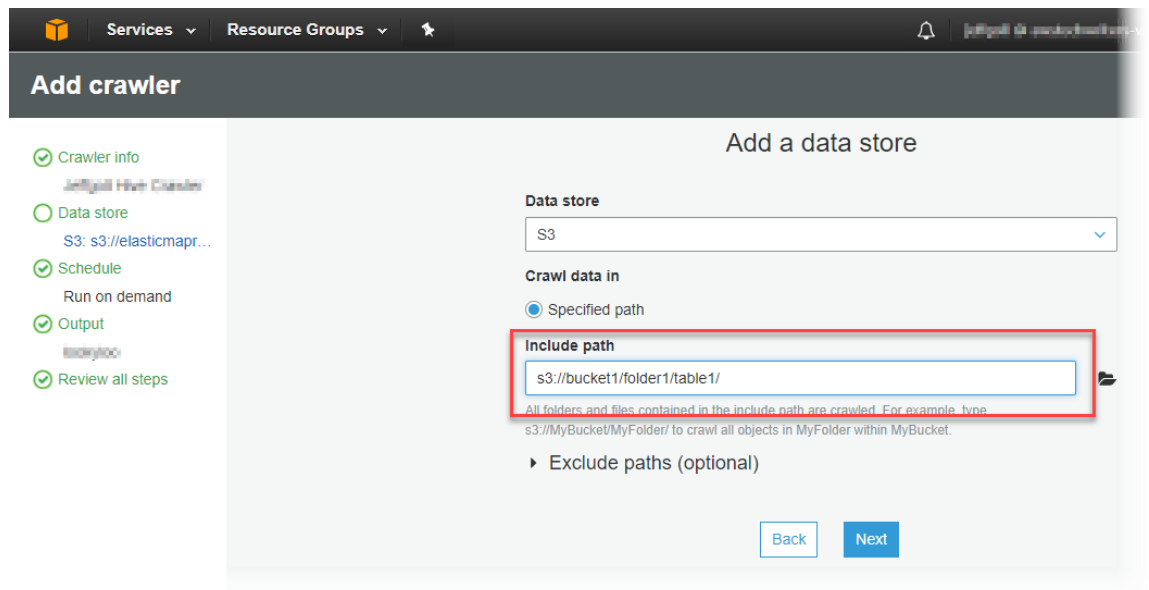
4. For **Add another data store**, choose **Yes**, **Next**.

5. 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

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. 25).

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

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

Updating Table Metadata

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

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

Working with CSV Files

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

CSV Data Enclosed in Quotes

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 OpenCSVSerDe. For more information about the OpenCSV SerDe, see OpenCSVSerDe for Processing CSV (p. 114).

In this case, you need to change the `serializationLib` property under field in the SerDeInfo field in the table to `org.apache.hadoop.hive.serde2.OpenCSVSerde` and enter appropriate values for `separatorChar`, `quoteChar`, and `escapeChar`.

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\""
```

The `separatorChar``value is a comma, the ``quoteChar` value is double quotes, and the `escapeChar` value is the backslash.

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": {
```

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

For more information, see Viewing and Editing Table Details in the *AWS Glue Developer Guide*.

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:

```
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 from within Athena must have a table property added to them called a `classification`, which identifies the format of the data. This allows AWS Glue to be able 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:

```
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, the property can be added using the AWS Glue console.

To change the classification property using the console

1. **Choose Edit Table.**
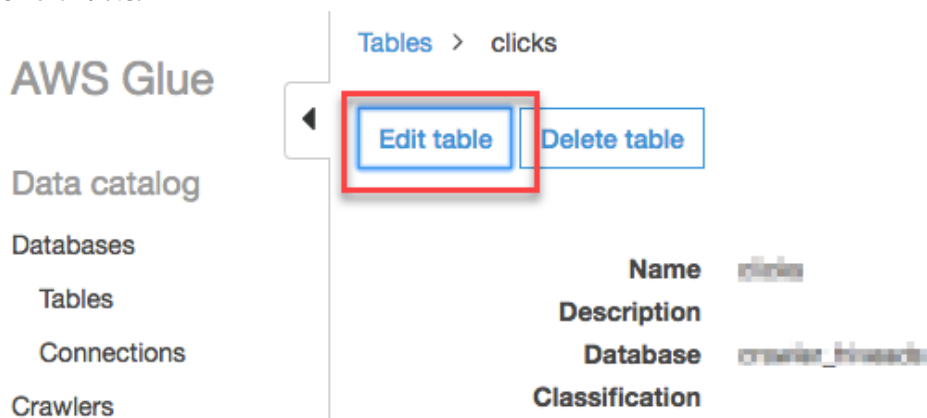
2. **For Classification, select the file type and choose Apply.**
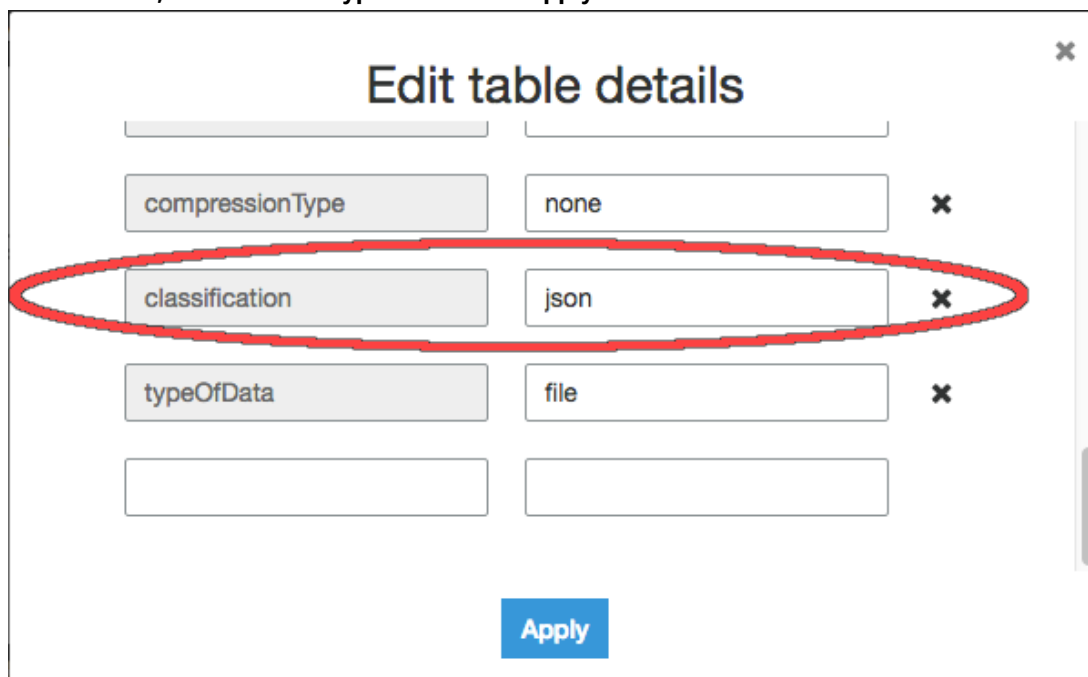
For more information, see Working with Tables in the *AWS Glue Developer Guide*.

Using ETL Jobs to Optimize Query Performance

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

We recommend the Parquet and ORC 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 Datatypes 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.

Changing Date Data Types to String for Parquet ETL Transformation

Athena currently does not support the `DATE` data type for Parquet files. Convert `DATE` data types to `STRING` when using the wizard or writing a script for an AWS Glue 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 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

You can use a JDBC connection to connect Athena to business intelligence tools, such as SQL Workbench. To do this, download, install, and configure the Athena JDBC driver, using the following link on Amazon S3.

Download the JDBC Driver

1. Download the driver (JDBC 4.1 and Java 8 compatible) from this location in Amazon S3 https://s3.amazonaws.com/athena-downloads/drivers/AthenaJDBC41-1.1.0.jar.
2. Use the AWS CLI with the following command:

```
aws s3 cp s3://athena-downloads/drivers/AthenaJDBC41-1.1.0.jar [local_directory]
```

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.

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/AthenaJDBC41-1.1.0.jar in the previous section. This makes the classes within the JAR available for use. The main JDBC driver class is `com.amazonaws.athena.jdbc.AthenaDriver`.

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

You must allow JDBC users to perform a set of policy-specific actions. These actions are not part of the Athena API. 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`

Configure the JDBC Driver Options

You can configure the following options for the JDBC driver. 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

Property Name	Description	Default Value	Is Required
`s3_staging_dir`	The S3 location to which your query output is written, for example `s3://query-results-bucket/folder/`, which is established under **Settings** in the Athena Console, https://console.aws.amazon.com/athena/. The JDBC driver then asks Athena to read the results and provide rows of data back to the user.	N/A	Yes
`query_results_encryption_option`	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`.	N/A	No
`query_results_aws_kms_key`	The key ID of the AWS customer master key (CMK) to use if `query_results_encryption_option`	N/A	No

Property Name	Description	Default Value	Is Required
	specifies `SSE-KMS` or `CSE-KMS`. For example, `123abcde-4e56-56f7-g890-1234h5678i9j`.		
`aws_credentials_provider_class`	The credentials provider class name, which implements the `AWSCredentialsProvider` interface.	N/A	No
`aws_credentials_provider_arguments`	Arguments for the credentials provider constructor as comma-separated values.	N/A	No
`max_error_retries`	The maximum number of retries that the JDBC client attempts to make a request to Athena.	10	No
`connection_timeout`	The maximum amount of time, in milliseconds, to make a successful connection to Athena before an attempt is terminated.	10,000	No
`socket_timeoutt`	The maximum amount of time, in milliseconds, to wait for a socket in order to send data to Athena.	10,000	No
`retry_base_delay`	Minimum delay amount, in milliseconds, between retrying attempts to connect Athena.	100	No
`retry_max_backoff_time`	Maximum delay amount, in milliseconds, between retrying attempts to connect to Athena.	1000	No
`log_path`	Local path of the Athena JDBC driver logs. If no log path is provided, then no log files are created.	N/A	No
`log_level`	Log level of the Athena JDBC driver logs. Valid values: `INFO`, `DEBUG`, `WARN`, `ERROR`, `ALL`, `OFF`, `FATAL`, `TRACE`.	N/A	No

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

- Windows 32 bit ODBC Driver
- Windows 64 bit ODBC Driver

Linux

- Linux 32 bit ODBC Driver
- Linux 64 bit ODBC Driver

OSX

ODBC Driver Connection String

For information about the connection string to use with your application, see ODBC Driver Installation and Configuration Guide.

Documentation

ODBC Driver Installation and Configuration Guide.

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

Setting User and Amazon S3 Bucket Permissions

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.

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

IAM Policies for User Access

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

Each 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.

The `AmazonAthenaFullAccess` policy is the managed policy for Athena. Attach this policy to users and other principals who need full access to Athena. 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*.

Customer-managed and *inline* policies allow you to specify more granular 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 allow additional actions not listed in the API. For more information, see Service Actions for JDBC Connections (p. 40).

AmazonAthenaFullAccess Managed Policy

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.

```
{
    "Version": "2012-10-17",
    "Statement": [
```

```
        {
            "Effect": "Allow",
            "Action": [
                "athena:*"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "glue:CreateDatabase",
                "glue:DeleteDatabase",
                "glue:GetDatabase",
                "glue:GetDatabases",
                "glue:UpdateDatabase",
                "glue:CreateTable",
                "glue:DeleteTable",
                "glue:BatchDeleteTable",
                "glue:UpdateTable",
                "glue:GetTable",
                "glue:GetTables",
                "glue:BatchCreatePartition",
                "glue:CreatePartition",
                "glue:DeletePartition",
                "glue:BatchDeletePartition",
                "glue:UpdatePartition",
                "glue:GetPartition",
                "glue:GetPartitions",
                "glue:BatchGetPartition"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetBucketLocation",
                "s3:GetObject",
                "s3:ListBucket",
                "s3:ListBucketMultipartUploads",
                "s3:ListMultipartUploadParts",
                "s3:AbortMultipartUpload",
                "s3:CreateBucket",
                "s3:PutObject"
            ],
            "Resource": [
                "arn:aws:s3:::aws-athena-query-results-*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::athena-examples*"
            ]
        }
    ]
}
```

AWSQuicksightAthenaAccess Managed Policy

An additional managed policy, `AWSQuicksightAthenaAccess`, grants access to actions that Amazon QuickSight needs to integrate with Athena. This policy includes deprecated actions for Athena that are not in the API. 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.

```
{
    "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: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",
                "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-*"
            ]
        }
    ]
}
```

Access through JDBC Connections

To gain access to AWS services and resources, such as Athena and the Amazon S3 buckets, provide JDBC driver credentials to your application. See Connect with the JDBC Driver (p. 33).

Amazon S3 Permissions

In addition to the allowed actions for Athena that you define in 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 user policies, bucket 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 this topic.

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

Cross-account Permissions

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.

```
{
    "Version": "2012-10-17",
    "Id": "MyPolicyID",
    "Statement": [
        {
            "Sid": "MyStatementSid",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::123456789123:root"
            },
```

```
        "Action": [
            "s3:GetBucketLocation",
            "s3:GetObject",
            "s3:ListBucket",
            "s3:ListBucketMultipartUploads",
            "s3:ListMultipartUploadParts",
            "s3:AbortMultipartUpload",
            "s3:PutObject"
        ],
        "Resource": [
            "arn:aws:s3:::my-athena-data-bucket",
            "arn:aws:s3:::my-athena-data-bucket/*"
        ]
    }
  ]
}
```

To grant access to a particular user in an account, replace the `Principal` key with a key that specifies the user instead of `root`. For example, for user profile `Dave`, use `arn:aws:iam::123456789123:user/Dave`.

Configuring Encryption Options

You can use Athena to query encrypted data in Amazon S3 by indicating data encryption when you create a table. You can also choose to 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. You set up query-result encryption using the Athena console or, if you connect using the JDBC driver, by configuring driver options. You specify the type of encryption to use and the Amazon S3 staging directory location. Query-result encryption applies to all queries.

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 SSL.

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.

Athena supports the following S3 encryption options, both for encrypted datasets in Amazon S3 and for encrypted query results:

- Server side encryption with an Amazon S3-managed key (SSE-S3)
- Server-side encryption with a AWS KMS-managed key (SSE-KMS).

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

- Client-side encryption with a AWS KMS-managed key (CSE-KMS)

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.

Permissions for Encrypting and Decrypting Data

If you use SSE-S3 for encryption, Athena users require no additional permissions for encryption and decryption. Having the appropriate Amazon S3 permissions for the appropriate Amazon S3 location (and for Athena actions) is enough. For more information about policies that allow appropriate Athena and Amazon S3 permissions, see IAM Policies for User Access (p. 37) and Amazon S3 Permissions (p. 40).

For data that is encrypted using AWS KMS, Athena users must be allowed to perform particular AWS KMS actions in addition to Athena and S3 permissions. You allow these actions by editing the key policy for the KMS customer master 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 KMS key policies. For information about how to add a user to a KMS key policy, see How to Modify a Key Policy in the *AWS Key Management Service Developer Guide*.

> **Note**
> Advanced key policy administrators may want to fine-tune 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*.

Creating Tables Based on Encrypted Datasets in Amazon S3

You indicate to Athena that a dataset is encrypted in Amazon S3 when you create a table (this is not required when using SSE-KMS). For both SSE-S3 and KMS encryption, Athena is able to determine 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.

> **Important**
> If you use Amazon EMR along with EMRFS to upload encrypted Parquet files, you must disable multipart uploads (set `fs.s3n.multipart.uploads.enabled` to `false`); otherwise, 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 *EMR Management Guide*.

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

- Use the CREATE TABLE (p. 136) statement with a `TBLPROPERTIES` clause that specifies `'has_encrypted_data'='true'`.
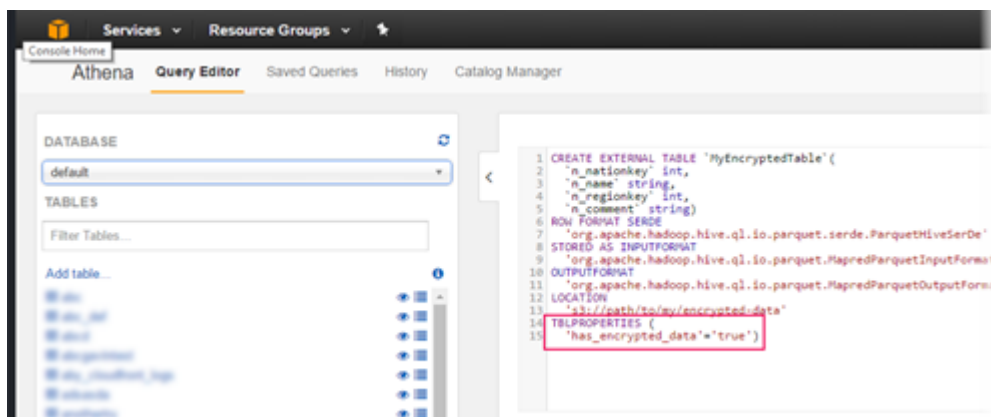
- Use the JDBC driver (p. 33) and set the `TBLPROPERTIES` value as above when you execute CREATE TABLE (p. 136) using `statement.executeQuery()`.

- 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**.
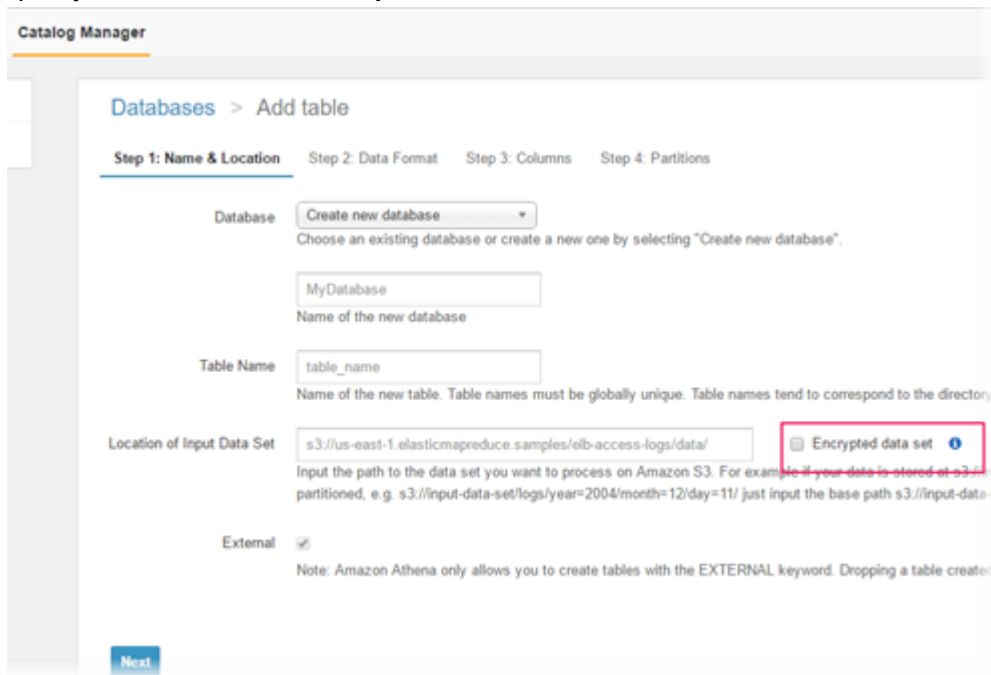
Tables based on encrypted data in Amazon S3 appear in the **Database** list with an encryption icon.
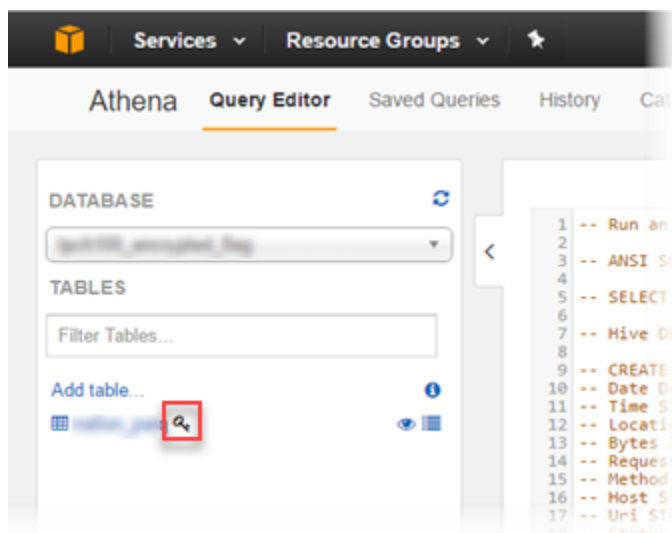
Encrypting Query Results Stored in Amazon S3

You use the Athena console or JDBC driver properties to specify that query results, which Athena stores in the S3 staging directory, are encrypted in Amazon S3. This setting applies to all Athena query results. You can't configure the setting for individual databases, tables, or queries.

To encrypt query results stored in Amazon S3 using the console

1. In the Athena console, choose **Settings**.
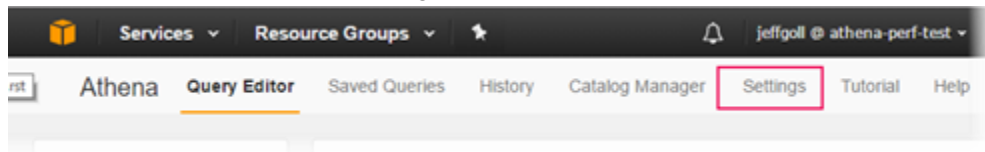
2. For **Query result location**, enter a custom value or leave the default. This is the Amazon S3 staging directory where query results are stored.

3. Choose **Encrypt query results**.

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

   If you chose **SSE-KMS** or **CSE-KMS**, for **Encryption key**, specify one of the following:

- If your account has access to an existing KMS CMK, choose its alias, or
- Choose **Enter a KMS key ARN** and then enter an ARN.
- To create a new KMS key, choose **Create KMS key**, use the IAM console to create the key, and then return to specify the key by alias or ARN as described in the previous steps. For more information, see Creating Keys in the *AWS Key Management Service Developer Guide*.

5. Choose **Save**.

Encrypting Query Results stored in Amazon S3 Using the JDBC Driver

You can configure the JDBC Driver to encrypt your query results using any of the encryption protocols that Athena supports. For more information, see JDBC Driver Options (p. 34).

Working with Source Data

Topics

Tables and Databases Creation Process in Athena

Amazon Athena uses Apache Hive data definition language (DDL) statements to define and query external tables where data resides on Amazon Simple Storage Service.

You can run DDL statements:

- In the Athena console
- using a JDBC or an ODBC driver
- 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.

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. 108).

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:

- 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. 37).

- If the data is not encrypted in Amazon S3, it can be stored n 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. 41).
- Athena does not support different storage classes within the bucket specified by the `LOCATION` clause, does not support the `GLACIER` storage class, and does not support Requester Pays buckets. For more information, see Storage Classes, Changing the Storage Class of an Object in Amazon S3, and Requester Pays Buckets in the *Amazon Simple Storage Service Developer Guide*.

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.

CREATE TABLE AS Type Statements Are Not Supported

Athena does not support CREATE TABLE AS type statements, for example, `CREATE TABLE AS SELECT`, which creates a table from the result of a SELECT query statement.

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. 148).

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

1. Open the Athena console at https://console.aws.amazon.com/athena/.
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. 24).
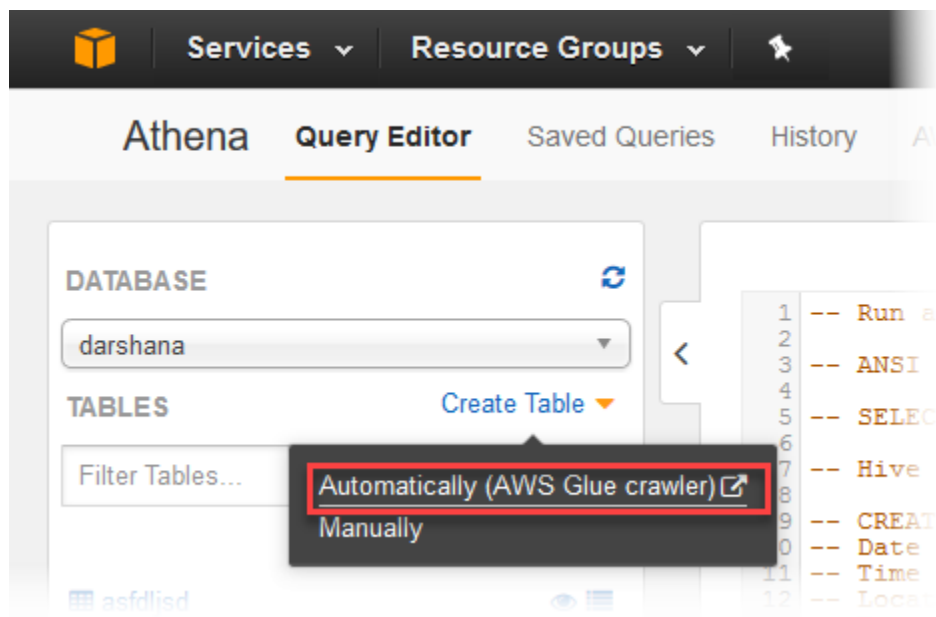
To create a table using the wizard

1. Open the Athena console at https://console.aws.amazon.com/athena/.
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.

1. Open the Athena console at https://console.aws.amazon.com/athena/.
2. Choose **Query Editor**.
3. Enter `CREATE DATABASE myDataBase` and choose **Run Query**.
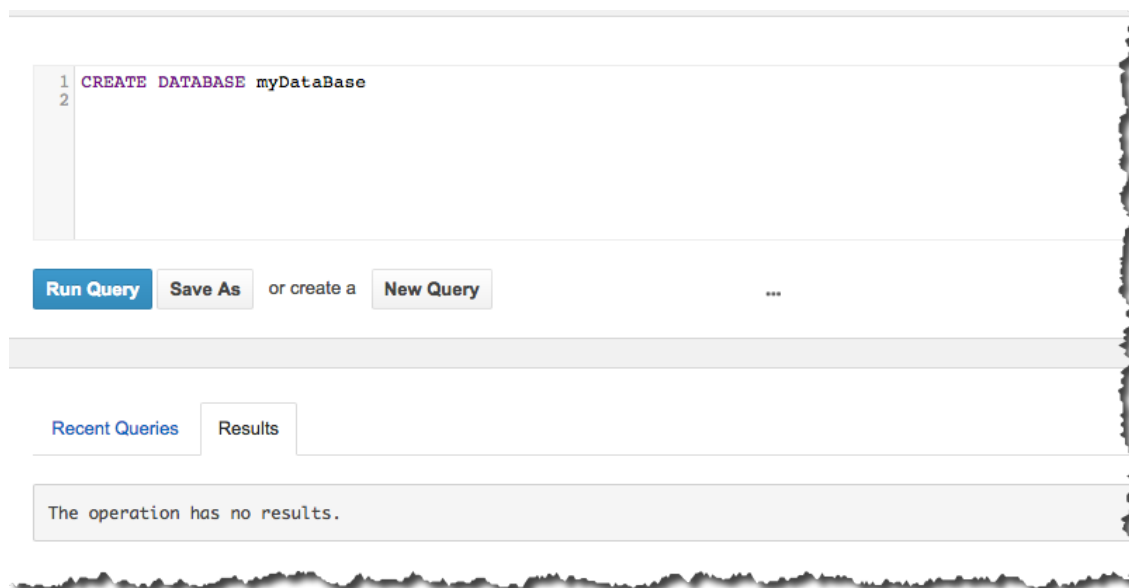
4. Select your database from the menu. It is likely to be an empty database.
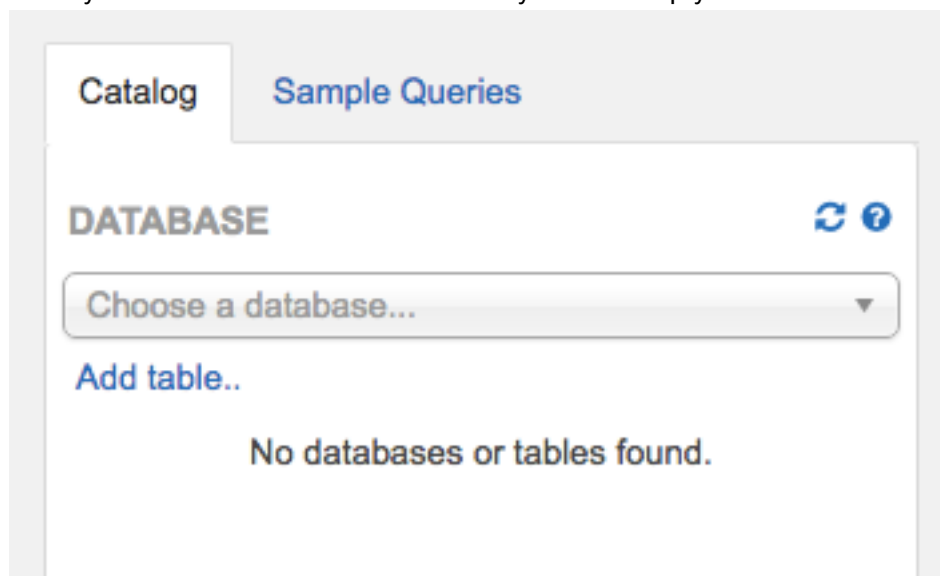
To create a table using Hive DDL

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**:

```
CREATE EXTERNAL TABLE IF NOT EXISTS cloudfront_logs (
    `Date` Date,
    Time STRING,
    Location STRING,
    Bytes INT,
    RequestIP STRING,
```

```
      Method STRING,
      Host STRING,
      Uri STRING,
      Status INT,
      Referrer STRING,
      OS String,
      Browser String,
      BrowserVersion String
) ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
"input.regex" = "^(?!#)([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s
+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s+[^\(]+[\(]([^\;]+).*\%20([^\/]+)[\/](.*)$"
) LOCATION 's3://athena-examples/cloudfront/plaintext/';
```

2. If the table was successfully created, you can then run queries against your data.

Names for Tables, Databases, and Columns

Use these tips for naming items in Athena.

Table names and table column names in Athena must be lowercase

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

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

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

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

Names that begin with an underscore

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

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

Table names that include numbers

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

```
CREATE TABLE "table123"
`_id` string,
`_index` string,
...
```

Table Location in Amazon S3

When you run a `CREATE TABLE AS` 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 folder or bucket.

Use:

```
s3://bucketname/keyname/
```

- Do not use filenames, wildcards, or glob patterns for specifying file locations.
- Do not add the full HTTP notation, such as `s3.amazon.com` to the Amazon S3 bucket path.

Do not use:

```
s3://path_to_bucket
s3://path_to_bucket/*
s3://path_to_bucket/mySpecialFile.dat
s3://bucketname/keyname/filename.csv
s3://test-bucket.s3.amazon.com
arn:aws:s3:::bucketname/keyname
```

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.

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.

Amazon Athena User Guide
Scenario 1: Data already partitioned
and stored on S3 in hive format

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

Storing Partitioned Data

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

```
aws s3 ls s3://elasticmapreduce/samples/hive-ads/tables/impressions/

    PRE dt=2009-04-12-13-00/
    PRE dt=2009-04-12-13-05/
    PRE dt=2009-04-12-13-10/
    PRE dt=2009-04-12-13-15/
    PRE dt=2009-04-12-13-20/
    PRE dt=2009-04-12-14-00/
    PRE dt=2009-04-12-14-05/
    PRE dt=2009-04-12-14-10/
    PRE dt=2009-04-12-14-15/
    PRE dt=2009-04-12-14-20/
    PRE dt=2009-04-12-15-00/
    PRE dt=2009-04-12-15-05/
```

Here, logs are stored with the column name (dt) set equal to date, hour, and minute increments. When you give a DDL with the location of the parent folder, the schema, and the name of the partitioned column, Athena can query data in those subfolders.

Creating a Table

To make a table out of this data, create a partition along 'dt' as in the following Athena DDL statement:

```
CREATE EXTERNAL TABLE impressions (
    requestBeginTime string,
    adId string,
    impressionId string,
    referrer string,
    userAgent string,
    userCookie string,
    ip string,
    number string,
    processId string,
    browserCookie string,
    requestEndTime string,
    timers struct<modelLookup:string, requestTime:string>,
    threadId string,
    hostname string,
    sessionId string)
PARTITIONED BY (dt string)
ROW FORMAT  serde 'org.apache.hive.hcatalog.data.JsonSerDe'
    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. 108).

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:

```
SELECT dt,impressionid FROM impressions WHERE dt<'2009-04-12-14-00' and
 dt>='2009-04-12-13-00' ORDER BY dt DESC LIMIT 100
```

This query should show you data similar to the following:

```
2009-04-12-13-20    ap3HcVKAWfXtgIPu6WpuUfAfL0DQEc
2009-04-12-13-20    17uchtodoS9kdeQP1x0XThKl5IuRsV
2009-04-12-13-20    JOUf1SCtRwviGw8sVcghqE5h0nkgtp
2009-04-12-13-20    NQ2XP0J0dvVbCXJ0pb4XvqJ5A4QxxH
2009-04-12-13-20    fFAItiBMsgqro9kRdIwbeX60SROaxr
2009-04-12-13-20    V4og4R9W6G3QjHHwF7gI1cSqig5D1G
2009-04-12-13-20    hPEPtBwk45msmwWTxPVVo1kVu4v11b
2009-04-12-13-20    v0SkfxegheD90gp31UCr6FplnKpx6i
2009-04-12-13-20    1iD9odVgOIi4QWkwHMcOhmwTkWDKfj
2009-04-12-13-20    b31tJiIA25CK8eDHQrHnbcknfSndUk
```

Scenario 2: Data is not partitioned

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

```
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
2016-11-23 17:54:46    9439710 elb/plaintext/2015/01/01/part-r-00005-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:47          0 elb/plaintext/2015/01/01_$folder$
2016-11-23 17:54:47    9012723 elb/plaintext/2015/01/02/part-r-00006-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:47    7571816 elb/plaintext/2015/01/02/part-r-00007-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:47    9673393 elb/plaintext/2015/01/02/part-r-00008-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:48   11979218 elb/plaintext/2015/01/02/part-r-00009-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:48    9546833 elb/plaintext/2015/01/02/part-r-00010-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:48   10960865 elb/plaintext/2015/01/02/part-r-00011-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:48          0 elb/plaintext/2015/01/02_$folder$
2016-11-23 17:54:48   11360522 elb/plaintext/2015/01/03/part-r-00012-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
```

```
2016-11-23 17:54:48    11211291 elb/plaintext/2015/01/03/part-r-00013-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:48     8633768 elb/plaintext/2015/01/03/part-r-00014-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:49    11891626 elb/plaintext/2015/01/03/part-r-00015-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:49     9173813 elb/plaintext/2015/01/03/part-r-00016-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:49    11899582 elb/plaintext/2015/01/03/part-r-00017-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:49           0 elb/plaintext/2015/01/03_$folder$
2016-11-23 17:54:50     8612843 elb/plaintext/2015/01/04/part-r-00018-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:50    10731284 elb/plaintext/2015/01/04/part-r-00019-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:50     9984735 elb/plaintext/2015/01/04/part-r-00020-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:50     9290089 elb/plaintext/2015/01/04/part-r-00021-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:50     7896339 elb/plaintext/2015/01/04/part-r-00022-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:51     8321364 elb/plaintext/2015/01/04/part-r-00023-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:51           0 elb/plaintext/2015/01/04_$folder$
2016-11-23 17:54:51     7641062 elb/plaintext/2015/01/05/part-r-00024-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:51    10253377 elb/plaintext/2015/01/05/part-r-00025-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:51     8502765 elb/plaintext/2015/01/05/part-r-00026-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:51    11518464 elb/plaintext/2015/01/05/part-r-00027-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:51     7945189 elb/plaintext/2015/01/05/part-r-00028-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:51     7864475 elb/plaintext/2015/01/05/part-r-00029-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:51           0 elb/plaintext/2015/01/05_$folder$
2016-11-23 17:54:51    11342140 elb/plaintext/2015/01/06/part-r-00030-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:51     8063755 elb/plaintext/2015/01/06/part-r-00031-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:52     9387508 elb/plaintext/2015/01/06/part-r-00032-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:52     9732343 elb/plaintext/2015/01/06/part-r-00033-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:52    11510326 elb/plaintext/2015/01/06/part-r-00034-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:52     9148117 elb/plaintext/2015/01/06/part-r-00035-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:52           0 elb/plaintext/2015/01/06_$folder$
2016-11-23 17:54:52     8402024 elb/plaintext/2015/01/07/part-r-00036-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:52     8282860 elb/plaintext/2015/01/07/part-r-00037-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:52    11575283 elb/plaintext/2015/01/07/part-r-00038-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:53     8149059 elb/plaintext/2015/01/07/part-r-00039-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:53    10037269 elb/plaintext/2015/01/07/part-r-00040-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:53    10019678 elb/plaintext/2015/01/07/part-r-00041-ce65fca5-d6c6-40e6-
b1f9-190cc4f93814.txt
2016-11-23 17:54:53           0 elb/plaintext/2015/01/07_$folder$
2016-11-23 17:54:53           0 elb/plaintext/2015/01_$folder$
2016-11-23 17:54:53           0 elb/plaintext/2015_$folder$
```

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 .

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.

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.

   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:

```
{
    "number": "977680",
    "referrer": "fastcompany.com",
    "processId": "1823",
    "adId": "TRktxshQXAHWo261jAHubijAoNlAqA",
    "browserCookie": "mvlrdwrmef",
    "userCookie": "emFlrLGrm5fA2xLFT5npwbPuG7kf6X",
    "requestEndTime": "1239714001000",
    "impressionId": "1I5G20RmOuG2rt7fFGFgsaWk9Xpkfb",
    "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": "bxxiuxduad",
    "hostname": "ec2-0-51-75-39.amazon.com",
    "sessionId": "J9NOccA3dDMFlixCuSOtl9QBbjs6aS",
    "requestBeginTime": "1239714000000"
}
```

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

```
MSCK REPAIR TABLE impressions;
```

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

```
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*:

```
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.

- Follow the instructions found in Setting Up (p. 12).

Example: Converting data to Parquet using an EMR cluster

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

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

```
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:

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

aws emr create-cluster
--applications Name=Hadoop Name=Hive Name=HCatalog \
--ec2-attributes KeyName=myKey,InstanceProfile=EMR_EC2_DefaultRole,SubnetId=subnet-
mySubnetId \
--service-role EMR_DefaultRole
--release-label emr-4.7.0
--instance-type \m4.large
--instance-count 1
--steps Type=HIVE,Name="Convert to Parquet",\
ActionOnFailure=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:

Querying Data in Amazon Athena Tables

Topics

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
- A metadata file (`*.csv.metadata`) that includes header information, such as column type.

If necessary, you can access the result files to work with them. Athena stores query results in this Amazon S3 bucket by default: `aws-athena-query-results-<ACCOUNTID>-<REGION>`.

To view or change the default location for saving query results, choose **Settings** in the upper right pane.

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

Query results are saved in an Amazon S3 location based on the name of the query and the date the query ran, as follows:

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

In this notation:

- `QueryLocation` is the base location for all query results. To view or change this location, choose **Settings**. You can enter a new value for **Query result location** at any time. You can also choose to encrypt query results in Amazon S3. For more information, see Configuring Encryption Options (p. 41).
- `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.

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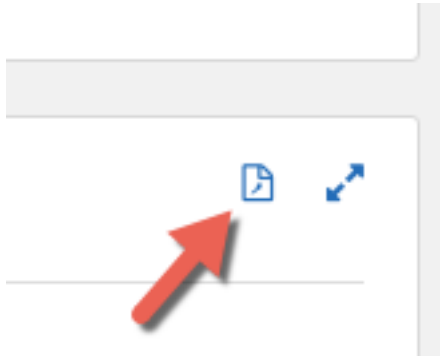
To save the results of a query you ran previously, choose **History**, locate your query, and use **Download Results**.

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 Java 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.

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

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:

```
+---------------------------------------+
| welcome_msg                           |
+---------------------------------------+
| [[hello, amazon, athena], [hi, alexa]] |
+---------------------------------------+
```

To create an array of key-value pairs, use the MAP operator that takes an array of keys followed by an array of values, as in this example:

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

This query returns:

```
+--------------------------------------------------------------------------------------------------------
+
| 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.

```
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.

```
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.

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

This query returns:

```
+-------+
| items |
+-------+
| [4,5] |
+-------+
```

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

```
SELECT
   ARRAY[CAST(MAP(ARRAY['a1', 'a2', 'a3'], ARRAY[1, 2, 3]) AS JSON)] ||
   ARRAY[CAST(MAP(ARRAY['b1', 'b2', 'b3'], ARRAY[4, 5, 6]) AS JSON)]
AS items
```

This query returns:

```
+-------------------------------------------------+
| items                                           |
+-------------------------------------------------+
| [{"a1":1,"a2":2,"a3":3}, {"b1":4,"b2":5,"b3":6}] |
+-------------------------------------------------+
```

Finding Lengths

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

```
SELECT cardinality(ARRAY[1,2,3,4]) AS item_count
```

This query returns:

```
+------------+
| item_count |
+------------+
| 4          |
+------------+
```

Accessing Array Elements

To access array elements, use the `[ ]` operator, with 1 specifying the first element, 2 specifying the second element, and so on, as in this example:

```
WITH dataset AS (
SELECT
    ARRAY[CAST(MAP(ARRAY['a1', 'a2', 'a3'], ARRAY[1, 2, 3]) AS JSON)] ||
    ARRAY[CAST(MAP(ARRAY['b1', 'b2', 'b3'], ARRAY[4, 5, 6]) AS JSON)]
AS items )
SELECT items[1] AS item FROM dataset
```

This query returns:

```
+-----------------------+
| item                  |
+-----------------------+
| {"a1":1,"a2":2,"a3":3} |
+-----------------------+
```

To access the elements of an array at a given position (known as the index position), use the `element_at()` function and specify the array name and the index position:

- If the index is greater than 0, `element_at()` returns the element that you specify, counting from the beginning to the end of the array. It behaves as the `[ ]` operator.
- If the index is less than 0, `element_at()` returns the element counting from the end to the beginning of the array.

The following query creates an array `words`, and selects the first element `hello` from it as the `first_word`, the second element `amazon` (counting from the end of the array) as the `middle_word`, and the third element `athena`, as the `last_word`.

```
WITH dataset AS (
  SELECT ARRAY ['hello', 'amazon', 'athena'] AS words
```

```
)
SELECT
  element_at(words, 1) AS first_word,
  element_at(words, -2) AS middle_word,
  element_at(words, cardinality(words)) AS last_word
FROM dataset
```

This query returns:

```
+----------------------------------------+
| first_word  | middle_word | last_word  |
+----------------------------------------+
| hello       | amazon      | athena     |
+----------------------------------------+
```

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:

```
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:

```
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:

```
+----------------------------------+
```

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

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:

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

Creating Arrays from Subqueries

Create an array from a collection of rows.

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

This query returns:

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

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

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

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

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

Filtering Arrays

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

```
WITH
dataset AS (
  SELECT ARRAY[1,2,3,4,5] AS items
)
SELECT array_agg(i) AS array_items
FROM dataset
CROSS JOIN UNNEST(items) AS t(i)
WHERE i > 3
```

This query returns:

```
+-------------+
| array_items |
+-------------+
| [4, 5]      |
+-------------+
```

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:

```
+--------------+
| array_items  |
```

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

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:

```
+--------------------+
| array_items        |
+--------------------+
| [1, 2, 3, 4, 5, 6] |
+--------------------+
```

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. 67).

```
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:

```
+--------------------+
| welcome_msg        |
+--------------------+
| hello amazon athena |
+--------------------+
```

Querying Arrays with ROWS and STRUCTS

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, order values, and find keywords using Athena queries.

- Creating a ROW (p. 70)
- Changing Field Names in Arrays Using CAST (p. 71)
- Filtering Arrays Using the . Notation (p. 71)
- Filtering Arrays with Nested Values (p. 72)
- Filtering Arrays Using UNNEST (p. 73)
- Finding Keywords in Arrays (p. 73)
- Ordering Values in Arrays (p. 74)

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 by Presto. When you use `CREATE_TABLE`, Athena defines a `STRUCT` in it, and relies on Presto for populating it with data. In turn, Presto creates the `ROW` data type for you, for each row in the dataset. The underlying `ROW` data type consists of named fields of any SQL data types supported in Presto.

```
WITH dataset AS (
 SELECT
   ROW('Bob', 38) AS users
 )
SELECT * FROM dataset
```

This query returns:

```
+------------------------+
```

```
| users                   |
+-------------------------+
| {field0=Bob, field1=38} |
+-------------------------+
```

Changing Field Names in Arrays Using `CAST`

To change the field name in an array that contains `ROW` values, you can `CAST` the `ROW` declaration:

```
WITH dataset AS (
  SELECT
    CAST(
      ROW('Bob', 38) AS ROW(name VARCHAR, age INTEGER)
    ) AS users
)
SELECT * FROM dataset
```

This query returns:

```
+-------------------+
| users             |
+-------------------+
| {NAME=Bob, AGE=38} |
+-------------------+
```

> **Note**
> In the example above, you declare `name` as a `VARCHAR` because this is its type in Presto. If you declare this `STRUCT` inside a `CREATE TABLE` statement, use `String` type because Hive defines this data type as `String`.

Filtering Arrays Using the `.` Notation

In the following example, select the `accountId` field from the `userIdentity` column of a AWS CloudTrail logs table by using the dot `.` notation. For more information, see Querying AWS CloudTrail Logs (p. 94).

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

This query returns:

```
+--------------+
| newid        |
+--------------+
| 112233445566 |
+--------------+
| 998877665544 |
+--------------+
```

To query an array of values, issue this query:

```
WITH dataset AS (
  SELECT ARRAY[
```

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

It returns this result:

```
+-----------------------------------------------------------------+
| users                                                           |
+-----------------------------------------------------------------+
| [{NAME=Bob, AGE=38}, {NAME=Alice, AGE=35}, {NAME=Jane, AGE=27}] |
+-----------------------------------------------------------------+
```

Filtering Arrays with Nested Values

Large arrays often contain nested structures, and you need to be able to filter, or search, for values within them.

To define a dataset for an array of values that includes a nested `BOOLEAN` value, issue this query:

```
WITH dataset AS (
  SELECT
    CAST(
      ROW('aws.amazon.com', ROW(true)) AS ROW(hostname VARCHAR, flaggedActivity ROW(isNew
 BOOLEAN))
    ) AS sites
)
SELECT * FROM dataset
```

It returns this result:

```
+----------------------------------------------------------+
| sites                                                    |
+----------------------------------------------------------+
| {HOSTNAME=aws.amazon.com, FLAGGEDACTIVITY={ISNEW=true}}  |
+----------------------------------------------------------+
```

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

```
WITH dataset AS (
  SELECT
    CAST(
      ROW('aws.amazon.com', ROW(true)) AS ROW(hostname VARCHAR, flaggedActivity ROW(isNew
 BOOLEAN))
    ) AS sites
)
SELECT sites.hostname, sites.flaggedactivity.isnew
FROM dataset
```

This query selects the nested fields and returns this result:

```
+-----------------------+
| hostname      | isnew |
+-----------------------+
| aws.amazon.com | true  |
```

```
+-----------------------+
```

Filtering Arrays Using UNNEST

To filter an array that includes a nested structure by one of its child elements, issue a query with an UNNEST operator. For more information about UNNEST, see .

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:

```
+-----------------------+
| hostname       | isnew |
+-----------------------+
| aws.amazon.com | true  |
+-----------------------+
```

Finding Keywords in Arrays

To search a dataset for a keyword within an element inside an array, use the regexp_like function.

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, use the regexp_like function.

Note
Instead of using the SQL LIKE operator, this query uses the regexp_like function. This function takes as an input a regular expression pattern to evaluate, or a list of terms separated by |, known as an OR operator. Additionally, searching for a large number of keywords is more efficient with the regexp_like function, because its performance exceeds that of the SQL LIKE operator.

```
WITH dataset AS (
  SELECT ARRAY[
    CAST(
      ROW('aws.amazon.com', ROW(ARRAY[
```

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

This query returns two sites:

```
+----------------+
| hostname       |
+----------------+
| aws.amazon.com |
+----------------+
| news.cnn.com   |
+----------------+
```

Ordering Values in Arrays

To order values in an array, use `ORDER BY`. The query in the following example adds up the total popularity scores for the sites matching your search terms with the `regexp_like` function, and then orders them from highest to lowest.

```
WITH dataset AS (
  SELECT ARRAY[
    CAST(
      ROW('aws.amazon.com', ROW(ARRAY[
          MAP(ARRAY['term', 'count'], ARRAY['bigdata', '10']),
          MAP(ARRAY['term', 'count'], ARRAY['serverless', '50']),
          MAP(ARRAY['term', 'count'], ARRAY['analytics', '82']),
          MAP(ARRAY['term', 'count'], ARRAY['iot', '74'])
      ])
        ) AS ROW(hostname VARCHAR, flaggedActivity ROW(flags ARRAY(MAP(VARCHAR, VARCHAR)) ))
```

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

This query returns this result:

```
+------------------------------------------------+
| hostname         | terms                | total |
+------------------------------------------------+
| news.cnn.com     | [politics,serverless] | 241   |
+------------------------------------------------+
| aws.amazon.com | [serverless]           | 10    |
+------------------------------------------------+
```

Querying Arrays with Maps

Maps are key-value pairs that consist of data types available in Athena.

To create maps, use the MAP operator and pass it two arrays: the first is the column (key) names, and the second is values. All values in the arrays must be of the same type. If any of the map value array elements need to be of different types, you can convert them later.

Examples

This example selects a user from a dataset. It uses the MAP operator and passes it two arrays. The first array includes values for column names, such as "first", "last", and "age". The second array consists of values for each of these columns, such as "Bob", "Smith", "35".

```
WITH dataset AS (
  SELECT MAP(
```

```
        ARRAY['first', 'last', 'age'],
        ARRAY['Bob', 'Smith', '35']
    ) AS user
)
SELECT user FROM dataset
```

This query returns:

```
+--------------------------------+
| user                           |
+--------------------------------+
| {last=Smith, first=Bob, age=35} |
+--------------------------------+
```

You can retrieve `Map` values by selecting the field name followed by `[key_name]`, as in this example:

```
WITH dataset AS (
 SELECT MAP(
    ARRAY['first', 'last', 'age'],
    ARRAY['Bob', 'Smith', '35']
 ) AS user
)
SELECT user['first'] AS first_name FROM dataset
```

This query returns:

```
+------------+
| first_name |
+------------+
| Bob        |
+------------+
```

Querying 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

Tips for Parsing Arrays with JSON

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 a JSON SerDe Libraries (p. 117) SerDe. 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. 117).

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. 77)
- Converting JSON to Athena Data Types (p. 77)

Converting Athena Data Types to JSON

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

```
WITH dataset AS (
  SELECT
    CAST('HELLO ATHENA' AS JSON) AS hello_msg,
    CAST(12345 AS JSON) AS some_int,
    CAST(MAP(ARRAY['a', 'b'], ARRAY[1,2]) AS JSON) AS some_map
)
SELECT * FROM dataset
```

This query returns:

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

Converting JSON to Athena Data Types

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

> **Note**
> In this example, to denote strings as JSON-encoded, start with the `JSON` keyword and use single quotes, such as `JSON '12345'`

```
WITH dataset AS (
  SELECT
    CAST(JSON '"HELLO ATHENA"' AS VARCHAR) AS hello_msg,
    CAST(JSON '12345' AS INTEGER) AS some_int,
    CAST(JSON '{"a":1,"b":2}' AS MAP(VARCHAR, INTEGER)) AS some_map
)
SELECT * FROM dataset
```

This query returns:

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

```
+-------------------------------------+
```

Extracting Data from JSON

You may have source data with containing JSON-encoded strings that you do not necessarily want to deserialize into a table in Athena. In this case, you can still run SQL operations on this data, using the JSON functions available in Presto.

Consider this JSON string as an example dataset.

```
{"name": "Susan Smith",
"org": "engineering",
"projects":
    [
     {"name":"project1", "completed":false},
     {"name":"project2", "completed":true}
    ]
}
```

Examples: extracting properties

To extract the `name` and `projects` properties from the JSON string, use the `json_extract` function as in the following example. The `json_extract` function takes the column containing the JSON string, and searches it using a `JSONPath`-like expression with the dot `.` notation.

> **Note**
> `JSONPath` performs a simple tree traversal. It uses the `$` sign to denote the root of the JSON document, followed by a period and an element nested directly under the root, such as `$.name`.

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

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

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

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

> **Note**
> Do not use the `json_extract_scalar` function on arrays, maps, or structs.

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

This query returns:

```
+---------------------------+
| name            | projects |
+---------------------------+
| Susan Smith     |          |
+---------------------------+
```

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

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

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

```
+------------------------------------+
| item                               |
+------------------------------------+
| {"name":"project1","completed":false} |
+------------------------------------+
```

To return an Athena string type, use the `[ ]` operator inside a `JSONPath` expression, then Use the `json_extract_scalar` function. For more information about `[ ]`, see Accessing Array Elements (p. 64).

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

It returns this result:

```
+--------------+
```

```
| project_name |
+--------------+
| project1     |
+--------------+
```

Searching for Values

To determine if a specific value exists inside a JSON-encoded array, use the `json_array_contains` function.

The following query lists the names of the users who are participating in "project2".

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

This query returns a list of users.

```
+-------------+
| user        |
+-------------+
| Susan Smith |
+-------------+
| Jane Smith  |
+-------------+
```

The following query example lists the names of users who have completed projects along with the total number of completed projects. It performs these actions:

- Uses nested `SELECT` statements for clarity.
- Extracts the array of projects.
- Converts the array to a native array of key-value pairs using `CAST`.
- Extracts each individual array element using the `UNNEST` operator.
- Filters obtained values by completed projects and counts them.

> **Note**
> When using `CAST` to `MAP` you can specify the key element as `VARCHAR` (native String in Presto), but leave the value as JSON, because the values in the `MAP` are of different types: String for the first key-value pair, and Boolean for the second.

```
WITH dataset AS (
  SELECT * FROM (VALUES
    (JSON '{"name": "Bob Smith",
            "org": "legal",
            "projects": [{"name":"project1", "completed":false}]}'),
    (JSON '{"name": "Susan Smith",
            "org": "engineering",
            "projects": [{"name":"project2", "completed":true},
                         {"name":"project3", "completed":true}]}'),
```

```
        (JSON '{"name": "Jane Smith",
                "org": "finance",
                "projects": [{"name":"project2", "completed":true}]}')
    ) AS t (users)
),
employees AS (
  SELECT users, CAST(json_extract(users, '$.projects') AS
    ARRAY(MAP(VARCHAR, JSON))) AS projects_array
  FROM dataset
),
names AS (
  SELECT json_extract_scalar(users, '$.name') AS name, projects
  FROM employees, UNNEST (projects_array) AS t(projects)
)
SELECT name, count(projects) AS completed_projects FROM names
WHERE cast(element_at(projects, 'completed') AS BOOLEAN) = true
GROUP BY name
```

This query returns the following result:

```
+---------------------------------+
| name          | completed_projects |
+---------------------------------+
| Susan Smith | 2               |
+---------------------------------+
| Jane Smith  | 1               |
+---------------------------------+
```

Obtaining Length and Size of JSON Arrays

Example: `json_array_length`

To obtain the length of a JSON-encoded array, use the `json_array_length` function.

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

This query returns this result:

```
+----------------------+
| name          | count |
```

```
+--------------------+
| Susan Smith | 2    |
+--------------------+
| Bob Smith   | 1    |
+--------------------+
| Jane Smith  | 1    |
+--------------------+
```

Example: `json_size`

To obtain the size of a JSON-encoded array or object, use the `json_size` function, and specify the column containing the JSON string and the `JSONPath` expression to the array or object.

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

This query returns this result:

```
+--------------------+
| name        | count |
+--------------------+
| Susan Smith | 2    |
+--------------------+
| Bob Smith   | 1    |
+--------------------+
| Jane Smith  | 1    |
+--------------------+
```

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?

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 pair of `double` type coordinates from the geographic coordinates of Mount Rainier in Athena, use the `ST_POINT (double, double) ((longitude, latitude) pair)` geospatial function, specifying the latitude and longitude:

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

Input Data Formats and Geometry Data Types

To use geospatial functions in Athena, input your data in the WKT or WKB formats, 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.
- **WKB (Well-known binary)**. In Athena, WKB is represented as a `varbinary` data type with a spatial reference ID (WKID) 4326. For information about both of these types, see the Wikipedia topic on Well-known text.
- **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. 117).

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.

Athena supports four types of geospatial functions:

- Constructor Functions (p. 85)
- Geospatial Relationship Functions (p. 86)
- Operation Functions (p. 88)
- Accessor Functions (p. 89)

Before You Begin

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

```
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 inlcuded 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 geospatial functions to obtain binary representations of a `point`, `` `line ``, or `polygon`. You can also convert a particular geometry data type to text, and obtain a binary representation of a geometry data type from text (WKT).

ST_POINT(double, double)

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

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 (0 0)')
FROM earthquakes
LIMIT 1;
```

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

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 1,0 -1,0 2))', 'POLYGON((-1 3,2 1,0 -3,-1 3))')
```

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

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

ST_CROSSES (geometry, geometry)

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

```
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 a geometry data type and a distance (as type double). Returns a binary representation of the geometry data type buffered by the specified distance.

Example:

```
SELECT ST_BUFFER(ST_Point(1, 2), 2.0)
```

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 one of the geometry data types and 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`, `multiline`, and `multipolygon`. For example, you can obtain an area of a `polygon` geometry data type, maximum and minimum X and Y values for a specified geometry data type, obtain the length of a `line`, or receive the number of points in a specified geometry data type.

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`. Example:

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

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 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 2)'))
```

ST_LENGTH (geometry)

Returns the length of `line` in type `double`. Example:

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

ST_MAX_X (geometry)

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

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

ST_MAX_Y (geometry)

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

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

ST_MIN_X (geometry)

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

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

ST_MIN_Y (geometry)

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

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

ST_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.

These examples rely on two files stored in an Amazon S3 location:

- 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 following example shows the `counties` table from this file with two fields only: `Name` (string), and `BoundaryShape` (binary).

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

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

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

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

```
CREATE external TABLE IF NOT EXISTS counties
 (
```

```
 Name string,
 BoundaryShape binary
 )
ROW FORMAT SERDE 'com.esri.hadoop.hive.serde.JsonSerde'
STORED AS INPUTFORMAT 'com.esri.json.hadoop.EnclosedJsonInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION 's3://my-query-log/json'
```

The following 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.

```
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:

```
+-----------------------+
| name            | cnt |
+-----------------------+
| Kern            | 36  |
+-----------------------+
| San Bernardino  | 35  |
+-----------------------+
| Imperial        | 28  |
+-----------------------+
| Inyo            | 20  |
+-----------------------+
| Los Angeles     | 18  |
+-----------------------+
| Riverside       | 14  |
+-----------------------+
| Monterey        | 14  |
+-----------------------+
| Santa Clara     | 12  |
+-----------------------+
| San Benito      | 11  |
+-----------------------+
| Fresno          | 11  |
+-----------------------+
| San Diego       | 7   |
+-----------------------+
| Santa Cruz      | 5   |
+-----------------------+
| Ventura         | 3   |
+-----------------------+
| San Luis Obispo | 3   |
+-----------------------+
| Orange          | 2   |
+-----------------------+
| San Mateo       | 1   |
+-----------------------+
```

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, and Amazon VPC flow 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 Accessing Amazon Athena with JDBC, 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. 12) and Getting Started (p. 14).

Topics

- Querying AWS CloudTrail Logs (p. 94)
- Querying Amazon CloudFront Logs (p. 97)
- Querying Classic Load Balancer Logs (p. 98)
- Querying Application Load Balancer Logs (p. 100)
- Querying Amazon VPC Flow Logs (p. 101)

Querying AWS CloudTrail Logs

AWS CloudTrail logs include details about any API calls made to your AWS services, including the console.

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. You can use Athena to query these log files directly from Amazon S3, specifying the `LOCATION` of log files. 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 attribute, such as source IP address or user.

A common application is to use CloudTrail logs to analyze operational activity for security and compliance. For a detailed example, see the AWS Big Data Blog post, Analyze Security, Compliance, and Operational Activity Using AWS CloudTrail and Amazon Athena.

CloudTrail saves logs as JSON text files in compressed gzip format (*.json.gzip). The location of log files depends on how you set up trails, the region or regions 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, as you need it in the next section.

Replace the `LOCATION` clause with the path to the CloudTrail log location and 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.

To query the CloudTrail logs, you use the CloudTrail SerDe (p. 112).

- Creating the Table for CloudTrail Logs (p. 95)
- Tips for Querying CloudTrail Logs (p. 96)

Creating the Table for CloudTrail Logs

To create the CloudTrail table

1. Copy and paste the following DDL statement into the Athena console.
2. Modify the `s3://cloudtrail_bucket_name/AWSLogs/bucket_ID/` to point to the S3 bucket that contains your logs data.

   Verify that fields are listed correctly. For a full list of fields in a CloudTrail record, see CloudTrail Record Contents.

   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. 78).

```
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/bucket_ID/';
```

3. Run the query in Athena console. After the query completes, Athena registers `cloudtrail_logs`, making the data in it ready for you to issue queries.

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 Creating the Table for CloudTrail Logs (p. 95). If it isn't the first table, delete the existing table: `DROP TABLE cloudtrail_logs;`.

- Recreate the table. For more information, see Creating the Table for CloudTrail Logs (p. 95).

  Verify that fields in your Athena query are listed correctly. For a 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. See Extracting Data From JSON (p. 78).

  Now you are ready to issue queries against your CloudTrail table.

- Start by looking at which IAM users called which API operations and from which source IP addresses.

- Use the following basic SQL query as your template. Paste the query to the Athena console and run it.

```
SELECT
```

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

- Modify this query further to explore your data.
- To improve performance and prevent long-running queries, include the `LIMIT` clause to return a specified subset of rows.

For more information, see the AWS Big Data blog post Analyze Security, Compliance, and Operational Activity Using AWS CloudTrail and Amazon Athena.

Querying Amazon CloudFront Logs

You can configure Amazon CloudFront CDN to export Web distribution access logs to Amazon Simple Storage Service. Use these logs to explore users' surfing patterns across your web properties served by CloudFront.

Before you begin querying the logs, enable Web distributions access log on your preferred CloudFront distribution. For information, see Access Logs in the *Amazon CloudFront Developer Guide*.

Make a note of the Amazon S3 bucket to which to save these logs.

> **Note**
> This procedure works for the Web distribution access logs in CloudFront. It does not apply to streaming logs from RTMP distributions.

- Creating the Table for CloudFront Logs (p. 97)
- Example Query for CloudFront logs (p. 98)

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 S3 bucket that stores your logs.

   This query uses the LazySimpleSerDe (p. 120) by default and it is omitted.

```
CREATE EXTERNAL TABLE IF NOT EXISTS default.cloudfront_logs (
  `date` DATE,
  time STRING,
  location STRING,
  bytes BIGINT,
  requestip STRING,
  method STRING,
  host STRING,
  uri STRING,
  status INT,
  referrer STRING,
  useragent STRING,
  querystring STRING,
  cookie STRING,
```

```
      resulttype STRING,
      requestid STRING,
      hostheader STRING,
      requestprotocol STRING,
      requestbytes BIGINT,
      timetaken FLOAT,
      xforwardedfor STRING,
      sslprotocol STRING,
      sslcipher STRING,
      responseresulttype STRING,
      httpversion STRING,
      filestatus STRING,
      encryptedfields INT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LOCATION 's3://your_log_bucket/prefix/'
TBLPROPERTIES ( 'skip.header.line.count'='2' )
```

2. Run the query in Athena console. After the query completes, Athena registers the `cloudfront_logs` table, making the data in it ready for you to issue queries.

Example Query for CloudFront logs

The following query adds up the number of bytes served by CloudFront between June 9 and June 11, 2017. 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 '2017-06-09' AND DATE '2017-06-11'
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 transferred.

Before you begin to 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.

Creating the Table for Elastic Load Balancing Logs

Create the table that you can later query. This table must include the exact location of your Elastic Load Balancing logs in Amazon S3.

To create the Elastic Load Balancing table

1. Copy and paste the following DDL statement into the Athena console.
2. Modify the `LOCATION` S3 bucket to specify the destination of your Elastic Load Balancing logs.

```
CREATE EXTERNAL TABLE IF NOT EXISTS elb_logs (
  request_timestamp string,
  elb_name string,
  request_ip string,
  request_port int,
  backend_ip string,
  backend_port int,
  request_processing_time double,
  backend_processing_time double,
  client_response_time double,
  elb_response_code string,
  backend_response_code string,
  received_bytes bigint,
  sent_bytes bigint,
  request_verb string,
  url string,
  protocol string,
  user_agent string,
  ssl_cipher string,
  ssl_protocol string
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
  'serialization.format' = '1',
  'input.regex' = '([^ ]*) ([^ ]*) ([^ ]*):([0-9]*) ([^ ]*)[:\-]([0-9]*) ([-.0-9]*)
  ([-.0-9]*) ([-.0-9]*) (|[-0-9]*) (-|[-0-9]*) ([-0-9]*) ([-0-9]*) \\\"([^ ]*) ([^ ]*) (-
  |[^ ]*)\\\" (\"[^\"]*\") ([A-Z0-9-]+) ([A-Za-z0-9.-]*)$' )
LOCATION 's3://your_log_bucket/prefix/AWSLogs/AWS_account_ID/elasticloadbalancing/';
```

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.

Example Queries for Elastic Load Balancing Logs

Use a query similar to this example. It lists the backend application servers that returned a `4XX` or `5XX` error response code. Use the `LIMIT` operator to limit the number of logs to query at a time.

```
SELECT
 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.

```
SELECT sum(backend_processing_time) AS
 total_ms,
 elb_name,
 backend_ip
```

```
FROM elb_logs WHERE backend_ip <> ''
GROUP BY backend_ip, elb_name
LIMIT 100;
```

For more information, see Analyzing Data in S3 using Athena.

Querying 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. 100)
- Example Queries for ALB logs (p. 101)

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_log_bucket/prefix/AWSLogs/your_ID/elasticloadbalancing/'`. For a full list of fields present in the ALB logs, see Access Log Entries.

   Create the `alb_logs` table as follows.

```
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
 )
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
'serialization.format' = '1',
```

```
'input.regex' = '([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*):([0-9]*) ([^ ]*)[:\-]([0-9]*) ([-.0-9]*)
 ([-.0-9]*) ([-.0-9]*) (|[-0-9]*) (-|[-0-9]*) ([-0-9]*) ([-0-9]*) \"([^ ]*) ([^ ]*) (- |
[^ ]*)\" (\"[^\"]*\") ([A-Z0-9-]+) ([A-Za-z0-9.-]*) ([^ ]*) (.*) (.*) (.*)' )
LOCATION 's3://your-alb-logs-directory/AWSLogs/elasticloadbalancing/';
```

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.

```
SELECT COUNT(request_verb) AS
 count,
 request_verb,
 client_ip
FROM alb_logs
GROUP BY request_verb, client_ip
LIMIT 100;
```

Another query shows the URLs visited by Safari browser users.

```
SELECT request_url
FROM alb_logs
WHERE user_agent LIKE '%Safari%'
LIMIT 10;
```

Querying Amazon VPC Flow Logs

Amazon Virtual Private Cloud flow logs capture information about the IP traffic going to and from network interfaces in a VPC. Use the logs to investigate network traffic patterns and identify threats and risks across your VPC network.

Before you begin querying the logs in Athena, enable VPC flow logs and export log data to Amazon S3. After you create the logs, let them run for a few minutes to collect some data.

- Creating the Table for VPC Flow Logs (p. 101)
- Example Queries for Amazon VPC Flow Logs (p. 102)

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.
2. Modify the `LOCATION 's3://your_log_bucket/prefix/'` to point to the S3 bucket that contains your log data.

```
CREATE EXTERNAL TABLE IF NOT EXISTS vpc_flow_logs (
  ts string,
  version int,
  account string,
  interfaceid string,
```

```
    sourceaddress string,
    destinationaddress string,
    sourceport int,
    destinationport int,
    protocol int,
    numpackets int,
    numbytes int,
    starttime string,
    endtime string,
    action string,
    logstatus string
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES
  ( "input.regex" = "^([^ ]+)\\s+([0-9]+)\\s+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\
\s+([0-9]+)\\s+([0-9]+)\\s+([0-9]+)\\s+([0-9]+)\\s+([0-9]+)\\s+([0-9]+)\\s+([0-9]+)\\s
+([^ ]+)\\s+([^ ]+)$" )
LOCATION 's3://your_log_bucket/prefix/';
```

3. Run the query in Athena console. After the query completes, Athena registers the `vpc_flow_logs` table, making the data in it ready for you to issue queries.

Example Queries for Amazon VPC Flow Logs

The following query lists all of the rejected TCP connections. The query uses a `datetime` function to convert the timestamp field `ts`, and extracts only the day of the week for which these events occurred.

```
SELECT day_of_week(from_iso8601_timestamp(ts)) AS
  day,
  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.

```
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.

Monitoring Logs and Troubleshooting

Topics

Logging Amazon Athena API Calls with AWS CloudTrail

Athena is integrated with CloudTrail, a service that captures all of the Athena API calls and delivers the log files to an Amazon S3 bucket that you specify. CloudTrail captures API calls from the Athena console or from your code to the Athena API operations. Using the information collected by CloudTrail, you can determine the request that was made to Athena, the source IP address from which the request was made, who made the request, when it was made, and so on.

You can also use Athena to query CloudTrail log files for insight. For more information, see CloudTrail SerDe (p. 112). To learn more about CloudTrail, including how to configure and enable it, see the AWS CloudTrail User Guide.

Athena Information in CloudTrail

When CloudTrail logging is enabled in your AWS account, API calls made to Athena actions are tracked in CloudTrail log files, where they are written with other AWS service records. CloudTrail determines when to create and write to a new file based on a time period and file size.

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 log entry contains information about who generated the request. The user identity information in the log entry helps you determine the following:

- Whether the request was made with root or 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 CloudTrail userIdentity Element in the *AWS CloudTrail User Guide*.

You can store your log files in your S3 bucket for as long as you want, but you can also define Amazon S3 lifecycle rules to archive or delete log files automatically. By default, your log files are encrypted with Amazon S3 server-side encryption (SSE).

To be notified upon log file delivery, you can configure CloudTrail to publish Amazon SNS notifications when new log files are delivered. For more information, see Configuring Amazon SNS Notifications for CloudTrail.

You can also aggregate Athena log files from multiple AWS regions and multiple AWS accounts into a single S3 bucket.

For more information, see Receiving CloudTrail Log Files from Multiple Regions and Receiving CloudTrail Log Files from Multiple Accounts.

Understanding Athena Log File Entries

CloudTrail log files can contain one or more log entries. Each entry lists multiple JSON-formatted events. A log entry 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. Log entries are not an ordered stack trace of the public API calls, so they do not appear in any specific order.

The following examples demonstrate CloudTrail log entries for:

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",
 "eventType":"AwsApiCall",
 "recipientAccountId":"123456789012"
}
```

StartQueryExecution (Failed)

```
{
 "eventVersion":"1.05",
 "userIdentity":{
  "type":"IAMUser",
  "principalId":"EXAMPLE_PRINCIPAL_ID",
  "arn":"arn:aws:iam::123456789012:user/johndoe",
```

```
  "accountId":"123456789012",
  "accessKeyId":"EXAMPLE_KEY_ID",
  "userName":"johndoe"
  },
 "eventTime":"2017-05-04T00:21:57Z",
 "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"
     },
  "requestID":"2487dd96-3a83-11e7-8f67-c9de5ac76512",
  "eventID":"15e3d3b5-6c3b-4c7c-bc0b-36a8dd95227b",
  "eventType":"AwsApiCall",
  "recipientAccountId":"123456789012"
},
```

Troubleshooting

Use these resources to troubleshoot problems with Amazon Athena.

- Service Limits (p. 158)
- 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

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 files in CSV, TSV, custom-delimited, and JSON formats; files from Hadoop-related formats: ORC, Apache Avro and Parquet; log files from Logstash, AWS CloudTrail logs, and Apache WebServer logs.

To create tables and query data from files 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 file formats.

Supported Data Formats and SerDes

Data Format	Description	SerDe types supported in Athena
CSV (Comma-Separated Values)	In a CSV file, 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. 120) if your data does not include values enclosed in quotes. • Use the OpenCSVSerDe for Processing CSV (p. 114) when your data includes quotes in values, or different separator or escape characters.
TSV (Tab-Separated Values)	In a TSV file, 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. 120) and specify the separator character as `FIELDS TERMINATED BY '\t'`.
Custom-Delimited files	In a file 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. 120) and specify custom delimiters.
JSON (JavaScript Object Notation)	In a JSON file, 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. 118). • Use the OpenX JSON SerDe (p. 118).
Apache Avro	A format for storing data in Hadoop that uses JSON-based schemas for record values.	Use the Avro SerDe (p. 109).
ORC (Optimized Row Columnar)	A format for optimized columnar storage of Hive data.	Use the ORC SerDe (p. 125) and ZLIB compression.

Data Format	Description	SerDe types supported in Athena
Apache Parquet	A format for columnar storage of data in Hadoop.	Use the Parquet SerDe (p. 127) and SNAPPY compression.
Logstash log files	A format for storing log files in Logstash.	Use the Grok SerDe (p. 115).
Apache WebServer log files	A format for storing log files in Apache WebServer.	Use the RegexSerDe for Processing Apache Web Server Logs (p. 111).
CloudTrail log files	A format for storing log files in CloudTrail.	• Use the CloudTrail SerDe (p. 112) to query most fields in CloudTrail logs. • Use the OpenX JSON SerDe (p. 118) for a few fields where their format depends on the service. For more information, see CloudTrail SerDe (p. 112).

Topics

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 an Avro file, you can 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 this 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",
        "type" : [ "null", "string" ],
        "default" : null
    }, {
        "name" : "origin",
        "type" : [ "null", "string" ],
```

```
      "default" : null
    }, {
      "name" : "dest",
      "type" : [ "null", "string" ],
      "default" : null
    }, {
      "name" : "depdelay",
      "type" : [ "null", "int" ],
      "default" : null
    }, {
      "name" : "carrierdelay",
      "type" : [ "null", "int" ],
      "default" : null
    }, {
      "name" : "weatherdelay",
      "type" : [ "null", "int" ],
      "default" : null
    } ]
}
')
STORED AS AVRO
LOCATION 's3://athena-examples-myregion/flight/avro/';
```

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

```
MSCK REPAIR TABLE flights_avro_example;
```

Query the top 10 departure cities by number of total departures.

```
SELECT origin, count(*) AS total_departures
FROM flights_avro_example
WHERE year >= '2000'
GROUP BY origin
ORDER BY total_departures DESC
LIMIT 10;
```

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

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/`.

```
CREATE EXTERNAL TABLE IF NOT EXISTS cloudfront_logs (
  `Date` DATE,
  Time STRING,
  Location STRING,
  Bytes INT,
  RequestIP STRING,
  Method STRING,
  Host STRING,
  Uri STRING,
  Status INT,
  Referrer STRING,
  os STRING,
  Browser STRING,
  BrowserVersion STRING
 ) ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
 WITH SERDEPROPERTIES (
 "input.regex" = "^(?!#)([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s
+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s+([^ ]+)\\s+[^\(]+[\(]([^\;]+).*\%20([^\/]+)[\/](.*)$"
 ) LOCATION 's3://athena-examples-myregion/cloudfront/plaintext/';
```

CloudTrail SerDe

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

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

In addition to using the CloudTrail SerDe, instances exist where you need to use a different SerDe or to extract data from JSON. Certain fields in CloudTrail logs are STRING values that may have a variable data format, which depends on the service. As a result, the CloudTrail SerDe is unable to predictably deserialize them. To query the following fields, identify the data pattern and then use a different SerDe, such as the OpenX JSON SerDe (p. 118). Alternatively, to get data out of these fields, use `JSON_EXTRACT` functions. For more information, see Extracting Data From JSON (p. 78).

- `requestParameters`
- `responseElements`
- `additionalEventData`
- `serviceEventDetails`

SerDe Name

`CloudTrail SerDe`

Library Name

`com.amazon.emr.hive.serde.CloudTrailSerde`

Examples

The following example uses the CloudTrail SerDe on a fictional set of log files 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. 78).

```
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/bucket_ID/';
```

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

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

```
        AND eventtime < '2017-02-18T00:00:00Z';
```

For more information, see Querying AWS CloudTrail Logs (p. 94).

OpenCSVSerDe for Processing CSV

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

- If the file contains values enclosed in quotes, use the OpenCSV SerDe to deserialize the values in Athena.
- If the file does not contain values enclosed in 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. 120).

CSV SerDe (OpenCSVSerde)

The OpenCSV SerDe behaves as follows:

- Allows you to specify separator, quote, and escape characters, such as: `WITH SERDEPROPERTIES ("separatorChar" = ",", "quoteChar" = "`", "escapeChar" = "\\")`
- Does not support embedded line breaks in CSV files.
- 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.

In particular, for data types other than `STRING` 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.fffffffff`" (9 decimal place precision). If you are processing CSV files 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 files from Hive, use the UNIX format for `DATE`.

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 a source CSV file saved in `s3://mybucket/mycsv/` with the following data contents:

```
"a1","a2","a3","a4"
"1","2","abc","def"
"a","a1","abc3","ab4"
```

Use a `CREATE TABLE` statement to create an Athena table based on this CSV file and reference the OpenCSVSerDe class in `ROW FORMAT`, also specifying SerDe properties for character separator, quote character, and escape character.

```
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 files, usually logs. Each Grok pattern is a named regular expression. You can identify and re-use these deserialization patterns as needed. This makes it easier to use Grok compared with using regular expressions. Grok provides a set of pre-defined patterns. You can also create custom patterns.

To specify the Grok SerDe when creating a table in Athena, use the `ROW FORMAT SERDE 'com.amazonaws.glue.serde.GrokSerDe'` clause, followed by the `WITH SERDEPROPERTIES` clause that specifies the patterns to match in your data, where:

- The `input.format` expression defines the patterns to match in the data file. 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.

- The `STORED AS INPUTFORMAT` and `OUTPUTFORMAT` clauses are required.
- The `LOCATION` clause specifies an Amazon S3 bucket, which can contain multiple source data files. All files 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 a single fictional text file saved in `s3://mybucket/groksample` with the following data, which represents Postfix maillog entries.

```
Feb  9 07:15:00 m4eastmail postfix/smtpd[19305]: B88C4120838: connect from
 unknown[192.168.55.4]
Feb  9 07:15:00 m4eastmail postfix/smtpd[20444]: B58C4330038: client=unknown[192.168.55.4]
Feb  9 07:15:03 m4eastmail postfix/cleanup[22835]: BDC22A77854: message-
id=<31221401257553.5004389LCBF@m4eastmail.example.com>
```

The following statement creates a table in Athena called `mygroktable` from the source data file, using a custom pattern and the predefined patterns that you specify.

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

Start with something simple like `%{NOTSPACE:column}` to get the columns mapped first and then specialize the columns if you want to.

Example 2

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

```
2017-09-12 12:10:34,972 INFO  - processType=AZ, processId=ABCDEFG614B6F5E49, status=RUN,
threadId=123:amqListenerContainerPool23[P:AJ|ABCDE9614B6F5E49||
2017-09-12T12:10:11.172-0700],
executionTime=7290, tenantId=12456, userId=123123f8535f8d76015374e7a1d87c3c,
 shard=testapp1,
jobId=12312345e5e7df0015e777fb2e03f3c, messageType=REAL_TIME_SYNC,
action=receive, hostname=1.abc.def.com
```

To query this log file:

- 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'
WITH SERDEPROPERTIES (
"input.grokCustomPatterns" = 'C_ACTION receive|send',
"input.format" = "%{TIMESTAMP_ISO8601:timestamp}, %{LOGLEVEL:loglevel} - processType=
%{NOTSPACE:processtype}, processId=%{NOTSPACE:processid}, status=%{NOTSPACE:status},
 threadId=%{NOTSPACE:threadid}, executionTime=%{POSINT:executiontime}, tenantId=
%{POSINT:tenantid}, userId=%{NOTSPACE:userid}, shard=%{NOTSPACE:shard}, jobId=
%{NOTSPACE:jobid}, messageType=%{NOTSPACE:messagetype}, action=%{C_ACTION:action},
 hostname=%{HOST:hostname}"
) STORED AS INPUTFORMAT 'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION 's3://mybucket /samples/';
```

JSON SerDe Libraries

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

- The native Hive JSON SerDe (p. 118)
- The OpenX JSON SerDe (p. 118)

This SerDe is used to process JSON documents, most commonly events. These events are represented as blocks of JSON-encoded text separated by a new line. The JsonSerDe is also capable of parsing more complex JSON documents with nested structures. However this requires a matching DDL representing the complex data types. There are only two properties for the JsonSerDe: `ignore.malformed.json`, which is self-explanatory, and `dots.in.keys`, which can be set to true or false and determines if there are dots in the name of the keys that will be replaced with underscores by the SerDe.

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

You can also use the Hive JSONSerDe to parse more complex JSON documents with nested structures. See Example: Deserializing Nested JSON (p. 119)

This SerDe has two useful optional properties you can specify when creating tables in Athena, to help deal with inconsistencies in the data:

- `'ignore.malformed.json'` if set to TRUE, lets you skip malformed JSON syntax.
- `'dots.in.keys'` if set to TRUE, specifies that the names of the keys include dots and replaces them with underscores.

> **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 JsonSerDe:

```
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 following DDL statement uses the OpenX 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.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 a nested JSON file. This example presumes a JSON file with the following structure:

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

The following `CREATE TABLE` command uses the Openx-JsonSerDe with collection data types like `struct` and `array` to establish groups of objects.

```
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 files 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. 120)
- TSV Example (p. 122)

> **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 CSV file 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,
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,
arrtime STRING,
arrdelay INT,
arrdelayminutes INT,
arrdel15 INT,
arrivaldelaygroups INT,
arrtimeblk STRING,
cancelled INT,
cancellationcode STRING,
diverted INT,
crselapsedtime INT,
actualelapsedtime INT,
airtime INT,
flights INT,
distance INT,
distancegroup INT,
carrierdelay INT,
weatherdelay INT,
nasdelay INT,
securitydelay INT,
lateaircraftdelay INT,
firstdeptime STRING,
totaladdgtime INT,
longestaddgtime INT,
divairportlandings INT,
divreacheddest INT,
divactualelapsedtime INT,
divarrdelay INT,
divdistance INT,
div1airport STRING,
div1airportid INT,
div1airportseqid INT,
div1wheelson STRING,
div1totalgtime INT,
div1longestgtime INT,
div1wheelsoff STRING,
div1tailnum STRING,
div2airport STRING,
```

```
        div2airportid INT,
        div2airportseqid INT,
        div2wheelson STRING,
        div2totalgtime INT,
        div2longestgtime INT,
        div2wheelsoff STRING,
        div2tailnum STRING,
        div3airport STRING,
        div3airportid INT,
        div3airportseqid INT,
        div3wheelson STRING,
        div3totalgtime INT,
        div3longestgtime INT,
        div3wheelsoff STRING,
        div3tailnum STRING,
        div4airport STRING,
        div4airportid INT,
        div4airportseqid INT,
        div4wheelson STRING,
        div4totalgtime INT,
        div4longestgtime INT,
        div4wheelsoff STRING,
        div4tailnum STRING,
        div5airport STRING,
        div5airportid INT,
        div5airportseqid INT,
        div5wheelson STRING,
        div5totalgtime INT,
        div5longestgtime INT,
        div5wheelsoff STRING,
        div5tailnum STRING
)
    PARTITIONED BY (year STRING)
    ROW FORMAT DELIMITED
      FIELDS TERMINATED BY ','
      ESCAPED BY '\\'
      LINES TERMINATED BY '\n'
    LOCATION 's3://athena-examples-myregion/flight/csv/';
```

Run `MSCK REPAIR TABLE` to refresh partition metadata each time a new partition is added to this table.

```
MSCK REPAIR TABLE flight_delays_csv;
```

Query the top 10 routes delayed by more than 1 hour.

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

TSV Example

This example presumes a source TSV file saved in `s3://mybucket/mytsv/`.

Use a `CREATE TABLE` statement to create an Athena table from the TSV file 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:

```
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,
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,
arrtime STRING,
arrdelay INT,
arrdelayminutes INT,
arrdel15 INT,
arrivaldelaygroups INT,
arrtimeblk STRING,
cancelled INT,
cancellationcode STRING,
diverted INT,
crselapsedtime INT,
actualelapsedtime INT,
airtime INT,
flights INT,
distance INT,
distancegroup INT,
carrierdelay INT,
weatherdelay INT,
nasdelay INT,
securitydelay INT,
lateaircraftdelay INT,
firstdeptime STRING,
totaladdgtime INT,
longestaddgtime INT,
divairportlandings INT,
divreacheddest INT,
```

```
  divactualelapsedtime INT,
  divarrdelay INT,
  divdistance INT,
  div1airport STRING,
  div1airportid INT,
  div1airportseqid INT,
  div1wheelson STRING,
  div1totalgtime INT,
  div1longestgtime INT,
  div1wheelsoff STRING,
  div1tailnum STRING,
  div2airport STRING,
  div2airportid INT,
  div2airportseqid INT,
  div2wheelson STRING,
  div2totalgtime INT,
  div2longestgtime INT,
  div2wheelsoff STRING,
  div2tailnum STRING,
  div3airport STRING,
  div3airportid INT,
  div3airportseqid INT,
  div3wheelson STRING,
  div3totalgtime INT,
  div3longestgtime INT,
  div3wheelsoff STRING,
  div3tailnum STRING,
  div4airport STRING,
  div4airportid INT,
  div4airportseqid INT,
  div4wheelson STRING,
  div4totalgtime INT,
  div4longestgtime INT,
  div4wheelsoff STRING,
  div4tailnum STRING,
  div5airport STRING,
  div5airportid INT,
  div5airportseqid INT,
  div5wheelson STRING,
  div5totalgtime INT,
  div5longestgtime INT,
  div5wheelsoff STRING,
  div5tailnum STRING
)
 PARTITIONED BY (year STRING)
 ROW FORMAT DELIMITED
  FIELDS TERMINATED BY '\t'
  ESCAPED BY '\\'
  LINES TERMINATED BY '\n'
 LOCATION 's3://athena-examples-myregion/flight/tsv/';
```

Run `MSCK REPAIR TABLE` to refresh partition metadata each time a new partition is added to this table.

```
MSCK REPAIR TABLE flight_delays_tsv;
```

Query the top 10 routes delayed by more than 1 hour.

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

Note

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

ORC SerDe

SerDe Name

OrcSerDe

Library Name

This is the SerDe class for ORC files. It passes the object from the ORC file to the reader and from the ORC file 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,
    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,
arrtime STRING,
arrdelay INT,
arrdelayminutes INT,
arrdel15 INT,
arrivaldelaygroups INT,
arrtimeblk STRING,
cancelled INT,
cancellationcode STRING,
diverted INT,
crselapsedtime INT,
actualelapsedtime INT,
airtime INT,
flights INT,
distance INT,
distancegroup INT,
carrierdelay INT,
weatherdelay INT,
nasdelay INT,
securitydelay INT,
lateaircraftdelay INT,
firstdeptime STRING,
totaladdgtime INT,
longestaddgtime INT,
divairportlandings INT,
divreacheddest INT,
divactualelapsedtime INT,
divarrdelay INT,
divdistance INT,
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_pq
WHERE depdelayminutes > 60
GROUP BY origin, dest
ORDER BY 3 DESC
LIMIT 10;
```

Parquet SerDe

SerDe Name

ParquetHiveSerDe is used for files stored in Parquet. For more information, see Parquet Format.

Library Name

Athena uses this class when it needs to deserialize files 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 `CREATE TABLE` statement to create an Athena table from the underlying CSV file stored in Amazon S3 in Parquet.

```
CREATE EXTERNAL TABLE flight_delays_pq (
    yr INT,
    quarter INT,
    month INT,
```

```
dayofmonth INT,
dayofweek INT,
flightdate STRING,
uniquecarrier STRING,
airlineid INT,
carrier STRING,
tailnum STRING,
flightnum STRING,
originairportid INT,
originairportseqid INT,
origincitymarketid INT,
origin STRING,
origincityname STRING,
originstate STRING,
originstatefips STRING,
originstatename STRING,
originwac INT,
destairportid INT,
destairportseqid INT,
destcitymarketid INT,
dest STRING,
destcityname STRING,
deststate STRING,
deststatefips STRING,
deststatename STRING,
destwac INT,
crsdeptime STRING,
deptime STRING,
depdelay INT,
depdelayminutes INT,
depdel15 INT,
departuredelaygroups INT,
deptimeblk STRING,
taxiout INT,
wheelsoff STRING,
wheelson STRING,
taxiin 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,
flights INT,
distance INT,
distancegroup INT,
carrierdelay INT,
weatherdelay INT,
nasdelay INT,
securitydelay INT,
lateaircraftdelay INT,
firstdeptime STRING,
totaladdgtime INT,
longestaddgtime INT,
divairportlandings INT,
divreacheddest INT,
divactualelapsedtime INT,
divarrdelay INT,
divdistance INT,
```

```
        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 PARQUET
LOCATION 's3://athena-examples-myregion/flight/parquet/'
tblproperties ("parquet.compress"="SNAPPY");
```

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

```
MSCK REPAIR TABLE flight_delays_pq;
```

Query the top 10 routes delayed by more than 1 hour.

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

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

Compression Formats

Athena supports the following compression formats:

- SNAPPY. This is the default compression format for files in the Parquet format. For Amazon Kinesis Firehose logs, SNAPPY compression is not supported. Use GZIP instead.
- ZLIB. This is the default compression format for files in the ORC format.
- GZIP
- LZO

DDL and SQL Reference

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

Topics

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. 136).

The field `col_name` specifies the name for each column in the table Athena creates, along with the column's data type. If `col_name` begins with an underscore, enclose it in backticks, for example `` `_mycolumn` ``.

List of Supported Data Types in Athena

The `data_type` value in the `col_name` field of `CREATE TABLE` can be any of the following:

- **primitive_type**
    - TINYINT
    - SMALLINT
    - INT
    - BIGINT
    - BOOLEAN
    - DOUBLE
    - FLOAT
    - STRING
    - TIMESTAMP
    - DECIMAL [(precision, scale)]
    - DATE (not supported for PARQUET file_format)
    - 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.
- **array_type**
    - ARRAY < data_type >
- **map_type**
    - MAP < primitive_type, data_type >

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

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. 148).

Topics

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. 51).

Synopsis

```
ALTER TABLE table_name ADD [IF NOT EXISTS]
  PARTITION
  (partition_col1_name = partition_col1_value
  [,partition_col2_name = partition_col2_value]
  [,...])
  [LOCATION 'location1']
  [PARTITION
  (partition_colA_name = partition_colA_value
  [,partition_colB_name = partition_colB_value
  [,...])]
  [LOCATION 'location2']
  [,...]
```

Parameters

[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 DATABASE IF NOT EXISTS clickstreams
  COMMENT 'Site Foo clickstream data aggregates'
  LOCATION 's3://myS3location/clickstreams'
  WITH DBPROPERTIES ('creator'='Jane D.', 'Dept.'='Marketing analytics');
```

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 `` `_mycolumn` ``. The `data_type` value can be any of the following:

- **primitive_type**
    - TINYINT
    - SMALLINT
    - INT
    - BIGINT
    - BOOLEAN
    - DOUBLE
    - FLOAT
    - STRING
    - TIMESTAMP
    - DECIMAL [(precision, scale)]
    - DATE (not supported for PARQUET file_format)
    - 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.
- **array_type**
    - ARRAY < data_type >
- **map_type**
    - MAP < primitive_type, data_type >

- **struct_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. 51).

> **Note**
> After you create a table with partitions, run a subsequent query that consists of the MSCK REPAIR TABLE (p. 141) 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. 46).

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

Use: `s3://mybucket/myfolder/`

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, `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. 41).

To run ETL jobs, AWS Glue requires that you create a table with the `classification` property
to indicate the data type for AWS Glue as `csv`, `parquet`, `orc`, `avro`, or `json`. For example,
`'classification'='csv'`. ETL jobs will fail if you do not specify this property. You can
subsequently specify it using the AWS Glue console, API, or CLI. For more information, see Using
AWS Glue Jobs for ETL with Athena (p. 30) and Authoring Jobs in 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+[^\(]+[\(]([^\;]+).*\%20([^\/]+)
[\/](.*)$"
      ) LOCATION 's3://athena-examples/cloudfront/plaintext/';
```

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;
```

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 [PURGE]
```

Parameters

[IF EXISTS]

Causes the error to be suppressed if `table_name` doesn't exist.

[PURGE]

Applies to managed tables. Ignored for external tables. Specifies that data should be removed permanently rather than being moved to the `.Trash/Current` directory.

Examples

```
DROP TABLE fulfilled_orders;
```

```
DROP TABLE IF EXISTS fulfilled_orders PURGE;
```

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. 51).

Synopsis

```
MSCK REPAIR TABLE table_name
```

Examples

```
MSCK REPAIR TABLE orders;
```

SHOW COLUMNS

Lists the columns in the table schema.

Synopsis

```
SHOW COLUMNS IN table_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.

Examples

```
SHOW CREATE TABLE orderclickstoday;
```

```
SHOW CREATE TABLE salesdata.orderclickstoday;
```

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.

Synopsis

```
SHOW PARTITIONS table_name
```

Examples

```
SHOW PARTITIONS clicks;
```

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 wildcards `*`, 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.

Examples

```
SHOW TBLPROPERTIES orders;
```

```
SHOW TBLPROPERTIES orders('comment');
```

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. 149).

- SELECT (p. 144)
- Logical Operators
- Comparison Functions and Operators
- Conditional Expressions
- Conversion Functions
- Mathematical Functions and Operators
- Bitwise Functions
- Decimal Functions and Operators
- String Functions and Operators
- Binary Functions
- Date and Time Functions and Operators
- Regular Expression Functions
- JSON Functions and Operators
- URL Functions
- Aggregate Functions
- Window Functions
- Color Functions
- Array Functions and Operators
- Map Functions and Operators
- Lambda Expressions and Functions
- Teradata Functions

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:

```
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. `SYTSTEM` 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. 59).

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
- ALTER TABLE `table_name` partitionSpec COMPACT
- ALTER TABLE `table_name` partitionSpec CONCATENATE
- ALTER TABLE `table_name` partitionSpec REPLACE COLUMNS
- ALTER TABLE `table_name` partitionSpec SET FILEFORMAT
- ALTER TABLE `table_name` RENAME TO
- ALTER TABLE `table_name` SET SKEWED LOCATION
- ALTER TABLE `table_name` SKEWED BY
- ALTER TABLE `table_name` TOUCH
- ALTER TABLE `table_name` UNARCHIVE PARTITION
- COMMIT
- CREATE INDEX
- CREATE ROLE
- CREATE TABLE `table_name` LIKE `existing_table_name`
- CREATE TEMPORARY MACRO
- CREATE VIEW
- DELETE FROM
- DESCRIBE DATABASE
- DFS
- DROP INDEX
- DROP ROLE
- DROP TEMPORARY MACRO
- EXPORT TABLE
- GRANT ROLE
- IMPORT TABLE
- INSERT INTO
- LOCK DATABASE
- LOCK TABLE
- REVOKE ROLE
- ROLLBACK
- SHOW COMPACTIONS
- SHOW CURRENT ROLES
- SHOW GRANT

- SHOW INDEXES
- SHOW LOCKS
- SHOW PRINCIPALS
- SHOW ROLE GRANT
- SHOW ROLES
- SHOW TRANSACTIONS
- START TRANSACTION
- UNLOCK DATABASE
- UNLOCK TABLE

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. 131).
- `CREATE TABLE AS SELECT` statements.
- `INSERT INTO` statements.
- 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.

Code Samples and Service Limits

Topics

Code Samples

Use examples in this topic as a starting point for writing Athena applications using the AWS SDK for Java.

> **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

```
package com.amazonaws.services.athena.sdksamples;

import com.amazonaws.ClientConfiguration;
import com.amazonaws.auth.InstanceProfileCredentialsProvider;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.athena.AmazonAthena;
import com.amazonaws.services.athena.AmazonAthenaClientBuilder;

/**
 * AthenaClientFactory
 * ------------------------------------
 * This code shows how to create and configure an Amazon Athena client.
 */
public class AthenaClientFactory
{
  /**
   * AmazonAthenaClientBuilder to build Athena with the following properties:
   * - Set the region of the client
   * - Use the instance profile from the EC2 instance as the credentials provider
   * - Configure the client to increase the execution timeout.
   */
  private final AmazonAthenaClientBuilder builder = AmazonAthenaClientBuilder.standard()
```

```
            .withRegion(Regions.US_EAST_1)
            .withCredentials(InstanceProfileCredentialsProvider.getInstance())
            .withClientConfiguration(new
 ClientConfiguration().withClientExecutionTimeout(ExampleConstants.CLIENT_EXECUTION_TIMEOUT));

  public AmazonAthena createClient()
  {
      return builder.build();
  }
}
```

Start Query Execution

```
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.
   */
  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()
```

```
                // You can provide encryption options for the output that is written.
                // .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.
  */

      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);
        }
 }

 /**
  * This code calls Athena and retrieves the results of a query.
  * The query must be in a completed state before the results can be retrieved and
  * paginated. The first row of results are the column headers.
  */
 private static void processResultRows(AmazonAthena client, String queryExecutionId)
 {
        GetQueryResultsRequest getQueryResultsRequest = new GetQueryResultsRequest()
                // Max Results can be set but if its not set,
                // it will choose the maximum page size
                // As of the writing of this code, the maximum value is 1000
                // .withMaxResults(1000)
                .withQueryExecutionId(queryExecutionId);
```

```
      GetQueryResultsResult getQueryResultsResult =
client.getQueryResults(getQueryResultsRequest);
      List<ColumnInfo> columnInfoList =
getQueryResultsResult.getResultSet().getResultSetMetadata().getColumnInfo();

      while (true) {
          List<Row> results = getQueryResultsResult.getResultSet().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:
                  throw new RuntimeException("Unexpected Type is not expected" +
columnInfoList.get(i).getType());
          }
      }
  }
}
```

Stop Query Execution

```
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_STAT
 {
          // 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)
  {
      QueryExecutionContext queryExecutionContext = new
 QueryExecutionContext().withDatabase(ExampleConstants.ATHENA_DEFAULT_DATABASE);

      ResultConfiguration resultConfiguration = new ResultConfiguration()
              .withOutputLocation(ExampleConstants.ATHENA_OUTPUT_BUCKET);
```

```
        StartQueryExecutionRequest startQueryExecutionRequest = new
 StartQueryExecutionRequest()
                .withQueryString(ExampleConstants.ATHENA_SAMPLE_QUERY)
                .withQueryExecutionContext(queryExecutionContext)
                .withResultConfiguration(resultConfiguration);

        StartQueryExecutionResult startQueryExecutionResult =
 client.startQueryExecution(startQueryExecutionRequest);

        return startQueryExecutionResult.getQueryExecutionId();
    }
}
```

List Query Executions

```
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(

 listQueryExecutionsRequest.withNextToken(listQueryExecutionsResult.getNextToken()));
          }
          else {
              hasMoreResults = false;
          }
      }
  }
```

```
}
```

Create a Named Query

```
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

```
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);

        // Process the results.
        boolean hasMoreResults = true;

        while (hasMoreResults) {
            List<String> namedQueryIds = listNamedQueriesResult.getNamedQueryIds();
            // process named query IDs
```

```
            // If nextToken is not null,  there are more results. Get the next page of
 results.
            if (listNamedQueriesResult.getNextToken() != null) {
                listNamedQueriesResult = client.listNamedQueries(

 listNamedQueriesRequest.withNextToken(listNamedQueriesResult.getNextToken()));
            }
            else {
                hasMoreResults = false;
            }
        }
    }
}
```

Service Limits

Note
You can contact AWS Support to request a limit increase for the limits listed here.

- By default, concurrency limits on your account allow you to submit five concurrent DDL queries (used for creating tables and adding partitions) and five concurrent SELECT queries at a time. This is a soft limit and you can request a limit increase for concurrent queries.
- If you use Athena in regions where AWS Glue is available, migrate to AWS Glue Catalog. See Upgrading to the AWS Glue Data Catalog Step-by-Step (p. 19). 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, 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.
- Query timeout: 30 minutes