Version 1.0.21

Querying Data by Using Athena

Lab overview and objectives

Sofía is happy with the proof of concept (POC) that you created, and the team is learning how to use it. However, one of the data scientists, Mary, would like to perform more advanced queries on comma-separated values (CSV) data. She often works with large datasets that are stored across multiple CSV files. She would like the ability to query multiple files in the same S3 bucket and aggregate the data from these files into the same database. She also wants to be able to transform the dataset metadata, such as column names and data type declarations. This information is part of the dataset's schema when the data is stored as a relational database.

Through some discovery, you learn that Amazon Athena provides this functionality. You explore Athena and its capabilities to see if you can address Mary's needs. Athena is an interactive query service that you can use to query data that is stored in Amazon S3. Athena stores data about the data sources that you query. You can store your queries for reuse, and you can share them with other users.

In this lab, you will learn how to use Athena and AWS Glue to query data that is stored in Amazon S3.

After completing this lab, you should be able to do the following:

- Use the Athena query editor to create an AWS Glue database and table.
- Define the schema for the AWS Glue database and associated tables by using the Athena bulk add columns feature.
- Configure Athena to use a dataset that is located in Amazon S3.
- Optimize Athena queries against a sample dataset.
- Create views in Athena to simplify data analysis for other users.
- Create Athena named queries by using AWS CloudFormation.
- Review an AWS Identity and Access Management (IAM) policy that can be assigned to users who intend to use Athena named queries.
- Confirm that a user can access an Athena named query by using the AWS Command Line Interface (AWS CLI) in the AWS Cloud9 terminal.

Duration

This lab will require approximately **90 minutes** to complete.

AWS service restrictions

In this lab environment, access to AWS services and service actions might be restricted to the ones that are needed to complete the lab instructions. You might encounter errors if you attempt to access other services or perform actions beyond the ones that are described in this lab.

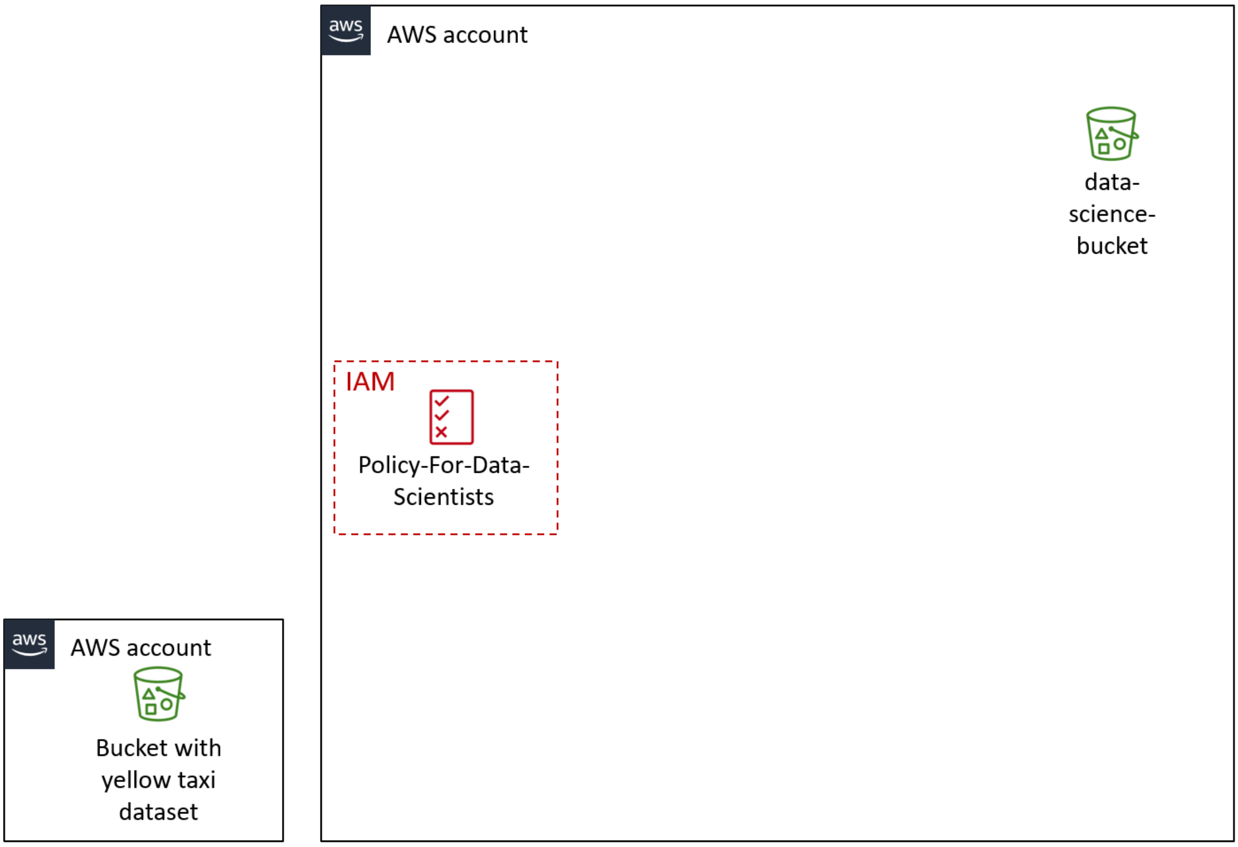
Scenario

In this lab, you will assume the role of a member of the data science team. You will build a POC using AWS Glue and Athena to analyze the data that's stored in Amazon S3. You want to experiment with Athena by accessing raw data in an S3 bucket, building an AWS Glue database, and querying this database by using Athena.

You also want to determine if you can scale this solution so that others on the team have access. Mary is part of the IAM group in AWS for the data science team and has similar access to the S3 bucket, AWS Glue database, and Athena. However, you limited her access to prevent unnecessary creation of resources. This follows the principle of least privilege, where an administrator limits access to AWS services and resources based upon what permissions are necessary to perform a specific job or task. In one task, you will use an IAM user to test the ability for a data science team member to run Athena managed queries. The IAM user was created for you, and the user belongs to an IAM group that has a policy attached to define permissions.

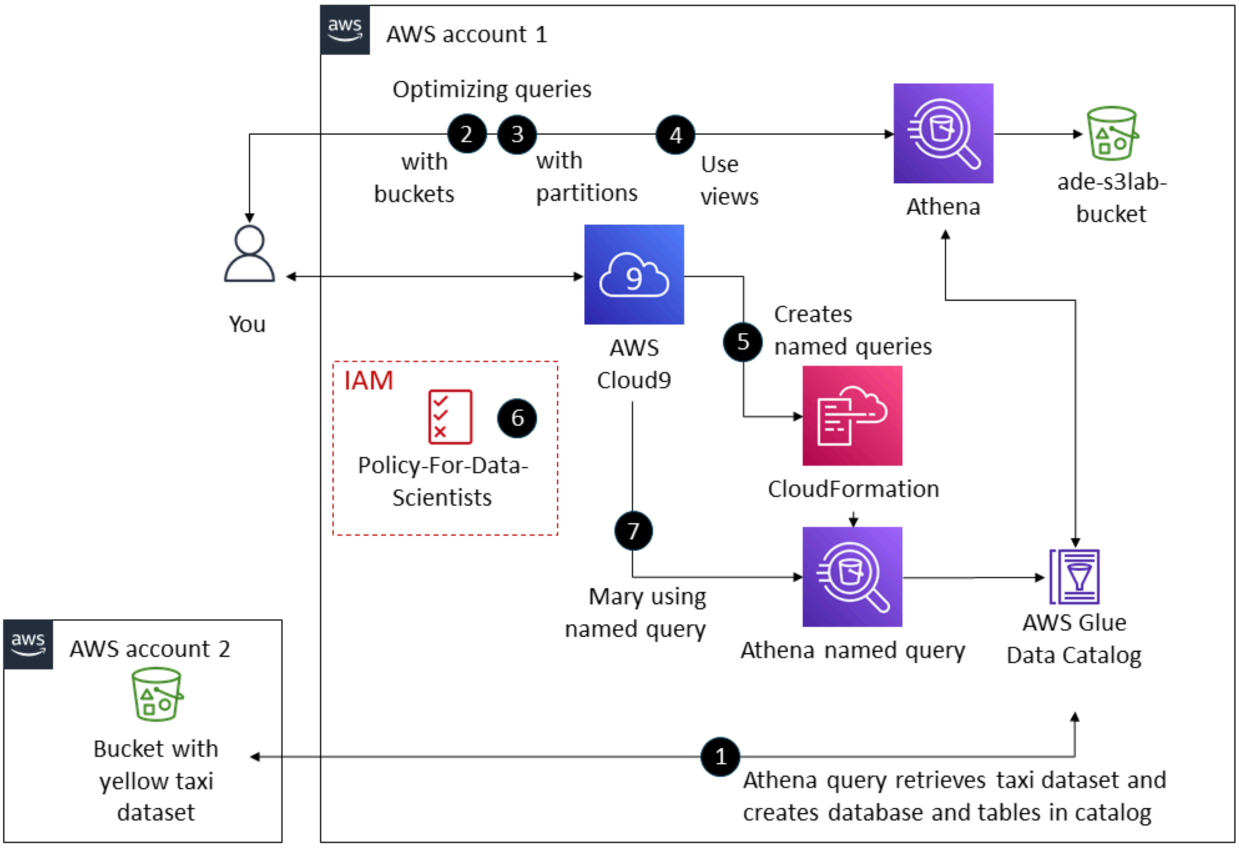
You ask Mary if she has a few sample datasets that you can use for your experiments. Mary provides access to a dataset that contains taxi trip data for 2017. You will use the data for your POC.

The following diagram illustrates the environment that was created for you in AWS at the *beginning* of the lab.



Tip: To review the CloudFormation template that built this environment, after logging in to the AWS Management Console, navigate to the CloudFormation console. In the navigation pane, choose **Stacks**.

By the *end* of the lab, you will have created the additional architecture that is shown in the following diagram. The table after the diagram provides a detailed explanation of the architecture in relation to the tasks that you will complete in this lab.



NUMBERED TASK	DETAIL
1	You will use Athena configuration steps and queries to create an AWS Glue database with data from the taxi dataset.
2	You will use buckets to analyze the taxi data for January. Note: The original dataset contains data for all months in the same file. Data for January has been split out of the original dataset into a separate file to optimize your Athena queries.
3	You will use Athena to analyze taxi data by using paytype to partition the data.
4	You will use Athena views to calculate the total value of fares paid for with credit card or cash.
5	You will compress the file by using the .zip and .gz file types and compare the sizes of the resulting files.
6	You will review the IAM policy called *Policy-For-Data-Scientists*.
7	Mary has the permissions from this policy because she is in the IAM group that the policy is assigned to. You will use this policy to test her access to the dataset and her ability to modify the original file properties within Amazon S3.

Accessing the AWS Management Console

1. At the top of these instructions, choose ▶ **Start Lab**.
   - The lab session starts.
   - A timer displays at the top of the page and shows the time remaining in the session.

     💡 **Tip:** To refresh the session length at any time, choose ▶ **Start Lab** again before the timer reaches 00:00.
   - Before you continue, wait until the circle icon to the right of the AWS ● link in the upper-left corner turns green.
2. To connect to the AWS Management Console, choose the **AWS** link in the upper-left corner.
   - A new browser tab opens and connects you to the console.

     💡 **Tip:** If a new browser tab does not open, a banner or icon is usually at the top of your browser with the message that your browser is preventing the site from opening pop-up windows. Choose the banner or icon, and then choose **Allow pop-ups**.

Task 1: Creating and querying an AWS Glue database and table in Athena

Your first task is to use SQL statements to define a schema for a table that can be used to work with the sample CSV data.

In this task, you will do the following:

- Specify an S3 bucket for query results.
- Create an AWS Glue database by using the Athena query editor.
- Create a table in the AWS Glue database and import data.
- Preview the data in the AWS Glue table.

4. In the AWS Management Console, in the search box to the right of **Services**, search for and choose **Athena** to open the Athena console.
5. Specify an S3 bucket for query results.

   When you use Athena, you must first specify an S3 bucket to hold the results for any queries that you run. A bucket was created for you. Complete the following steps to configure and use it.

   - In the Athena console, choose **Explore the query editor**.
   - Choose the **Settings** tab.
   - In the **Query result and encryption settings** section, choose **Manage**.
   - For **Location of query result**, choose **Browse S3**.
   - Choose the bucket that was created for you, and then select **Choose**.
   - Keep the defaults for the other settings on the page, and choose **Save**.

     Note: At this time, you aren't required to specify an **Expected bucket owner**.
6. Create an AWS Glue database by using the Athena query editor.
   - Choose the **Editor** tab.
   - In the **Query 1** section, enter the following SQL command:

     ```
     CREATE DATABASE taxidata;
     ```

   - Choose **Run**.

     The message *Query successful* displays, and an AWS Glue database named *taxidata* is created.

     Tip: To confirm that the database was created in AWS Glue, open a new tab and navigate to the AWS Glue console. In the navigation pane, choose **Databases**. The *taxidata* database is listed on the page.

Next, you will create a table in the AWS Glue database and import data.

7. In the Athena console, in the **Tables and views** section on the left, choose **Create** > **S3 bucket data**, and configure the following:
   - **Table name:** Enter `yellow`
   - **Description:** Enter `Table for taxi data`
   - **Database configuration:** Select **Choose an existing database**, and then choose **taxidata** from the dropdown list.
   - **Location of input dataset:** Copy the following link into the field:

```
s3://aws-tc-largeobjects/CUR-TF-200-ACDSCI-1/Lab2/yellow/
```

- **Encryption:** Keep the default setting, which is not selected.
- **Data format:** Choose **CSV**.
- In the **Column details** section, choose **Bulk add columns**.

  Note: This feature provides the ability to quickly add metadata to the table, such as column names and data type declarations.

- In the pop-up box, copy and paste the following text, and then choose **Add**:

```
vendor string,
pickup timestamp,
dropoff timestamp,
count int,
distance int,
ratecode string,
storeflag string,
pulocid string,
dolocid string,
paytype string,
fare decimal,
extra decimal,
mta_tax decimal,
tip decimal,
tolls decimal,
surcharge decimal,
total decimal
```

  The column names and data types populate in the **Column details** section.

- In the **Preview table query** section, view the preview table query text, which matches the following text:

```
CREATE EXTERNAL TABLE IF NOT EXISTS taxidata.yellow
(
    `vendor` string,
    `pickup` timestamp,
    `dropoff` timestamp,
    `count` int,
    `distance` int,
    `ratecode` string,
    `storeflag` string,
    `pulocid` string,
    `dolocid` string,
    `paytype` string,
    `fare` decimal,
    `extra` decimal,
    `mta_tax` decimal,
    `tip` decimal,
    `tolls` decimal,
    `surcharge` decimal,
    `total` decimal
  )
  ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'
  WITH SERDEPROPERTIES (
    'serialization.format' = ',',
    'field.delim' = ','
  ) LOCATION 's3://aws-tc-largeobjects/CUR-TF-200-ACDSCI-1/Lab2/yellow/'
  TBLPROPERTIES ('has_encrypted_data'='false');
```

Note that the data is not encrypted. Serde is a data serialization format, and the SERDE property specifies that the data must be comma delimited, which it is.

- Choose **Create table**.

  The message *Query successful* displays.

Now that you have a table with the taxi data, you can write queries to retrieve data from the data source in Amazon S3.

8. Preview data in the AWS Glue table.

   - In the **Data** section on the left, for **Database**, choose **taxidata**.
   - In the **Tables** section, to the right of the *yellow* table, choose the ellipsis (three dot) icon, and then choose **Preview Table**.

     The **Results** section displays the first 10 records of the table.
   - Before continuing, at the top of the query editor, close the open queries by choosing the X icon on each query tab. When prompted, choose **Close query**.

     Note: You can only have 10 active queries in Athena at any given time, so you must manage these accordingly.

Task 1 summary

In this task, you created an AWS Glue database and table by using the Athena query editor. You connected an Amazon S3 dataset to the table and defined the schema for the table by using the bulk add columns feature. After you created the table, you learned how to preview the data by using the preview table feature.

Mary can now use more advanced SQL queries and custom column data types for CSV files that are stored securely in Amazon S3.

Task 2: Optimizing Athena queries by using buckets

When you work with large datasets that are spread out across multiple files, two major goals are to optimize query performance and to minimize cost. Cost for Athena is based on usage, which is the amount of data that is scanned, and prices vary based on your Region.

Three possible strategies that you can use to minimize your costs and improve performance are compressing data, bucketizing data, and partitioning data.

- **Compressing data:** Compress your data by using one of the open standards for file compression (such as Gzip or tar). Compression results in a smaller size for the dataset when it's stored in Amazon S3.
- The cardinality of your data also effects how you should optimize your queries. For more information, see Cardinality (SQL Statements). There are two options to optimize based upon high or low cardinality. These are:
  - **Bucketizing data:** For high cardinality with data, store records in distinct *buckets* (not to be confused with S3 buckets) based upon a shared value in a specific field. Consider bucketizing data as part of the preprocessing phase of your data pipeline. In this lab, data for a single month will be bucketized separately from the original dataset, which contains data for an entire year. This strategy will help to optimize performance.
  - **Partitioning data:** You can also use partitions to improve performance and reduce cost. Partitioning is used with low-cardinality data, which means that fields have few unique or distinct values.

In this task, you will experiment with bucketizing data to optimize Athena queries. You will complete the following actions:

- Create a table named *jan* to hold bucketized data.
- Compare how long it takes to run a query against bucketized data for January 2017 and how long it takes to query the entire dataset for the January 2017 data.

9. Create a table for the January 2017 data.
    - In the Athena query editor, choose the **Editor** tab.
    - To open a new query tab, choose the plus icon on the right side of the query section.
    - Copy and paste the following text into the new query tab, and then choose **Run**:

```
CREATE EXTERNAL TABLE IF NOT EXISTS jan (
  `vendor` string,
  `pickup` timestamp,
  `dropoff` timestamp,
  `count` int,
  `distance` int,
  `ratecode` string,
  `storeflag` string,
  `pulocid` string,
  `dolocid` string,
  `paytype` string,
  `fare` decimal,
  `extra` decimal,
  `mta_tax` decimal,
  `tip` decimal,
  `tolls` decimal,
  `surcharge` decimal,
  `total` decimal
  )
  ROW FORMAT SERDE
'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'
  WITH SERDEPROPERTIES (
  'serialization.format' = ',',
  'field.delim' = ','
) LOCATION 's3://aws-tc-largeobjects/CUR-TF-200-
ACDSCI-1/Lab2/January2017/'
  TBLPROPERTIES ('has_encrypted_data'='false');
```

The message *Query successful* displays. Athena created a new table named *jan*, which contains the taxi data for only the month of January 2017. The table is listed in the **Tables** section on the left.

Analysis: In this step, you created a table to store data for only January 2017 by using Serde. The fares in this dataset all have *pickup* timestamps, and almost all the timestamps are unique for each record (meaning the records have high cardinality), but also have *January* in the *month* column, so you can use bucketizing. The january data is stored in a separate file in Amazon S3 from the file with the full dataset. In the following steps, you will investigate whether bucketizing these records in a separate file improves performance compared to querying for January records in the full dataset.

10. Run the following query on the *yellow* table, which has data for the entire year. The data is not divided into monthly buckets.

```
SELECT count (count) AS "Number of trips" ,
       sum (total) AS "Total fares" ,
       pickup AS "Trip date"
FROM yellow WHERE pickup

between TIMESTAMP '2017-01-01 00:00:00'
   and TIMESTAMP '2017-02-01 00:00:01'
GROUP BY pickup;
```

The message *Query successful* displays. Note the run time and amount of data scanned for the query.

Now, you will compare that by running a query against the *jan* table.

11. Run the following query on the *jan* table.

```
SELECT count (count) AS "Number of trips" ,
       sum (total) AS "Total fares" ,
       pickup AS "Trip date"
FROM jan
GROUP BY pickup;
```

Note the run time and amount of data scanned for the query. As you can see, much less data is scanned when the query is performed against the table that only contains the January data.

Analysis: Putting data in separate buckets works when you have data that has a high degree of cardinality. *Cardinality* refers to the number of distinct records in a database. In this example, the *pickup* field has a high degree of cardinality because every trip has a specific date and time.

Task 2 summary

In this task, you created a table for the January 2017 data. You learned how to optimize queries by using various strategies, such as dividing data into multiple buckets and partitioning data.

To accomplish this task, you first ran a query on data that was not divided into buckets and used this query as the baseline. Then, you ran a query on the January 2017 data, which was divided into a separate bucket, and compared the results to the baseline.

Task 3: Optimizing Athena queries by using partitions

If you are interested in querying a field with low cardinality, which means that the field has few unique values, you would partition the data instead of using distinct buckets. In some cases, your data will be partitioned by another process. To find the most efficient approach, you will try to partition the data by using the *paytype* field in a specific query in Athena.

The *paytype* field stores the type of payment by using the following codes:

- 1 = Credit card
- 2 = Cash
- 3 = No charge
- 4 = Dispute
- 5 = Unknown
- 6 = Voided trip

Because the set of possible values is limited, *paytype* is an excellent column to use to create partitions. In this task, you will use the CREATE TABLE AS function to partition the data. You will also specify a columnar storage format.

You can store data in Athena with the Apache Parquet or Optimized Row Columnar (ORC) formats. Columnar storage formats compress the data, which will further reduce costs for your queries.

In this task, you will experiment with using partitions with your queries. You will complete the following steps:

- Create a table that is based on the Apache Parquet format and uses the *paytype* field as a partition.
- Compare the time it takes to run a query against the *yellow* database for records with paytype 1 (credit card) to how long it takes to query the partitioned table that was built with the Apache Parquet format.

To begin, you will create a partition by running a query to select the data that you want to use for a partition and to indicate a storage format.

12. To create a new table called *taxidata.creditcard* that is partitioned for *paytype = 1* (credit card transactions), run the following query in a new query tab:

    ```
    CREATE TABLE taxidata.creditcard
    WITH (
      format = 'PARQUET'
      )AS
      SELECT * from "yellow"
      WHERE paytype = '1';
    ```

    The run time and data scanned values are similar to the following:

    ```
    Time in queue: 251 ms
    Run time: 28.572 sec
    Data scanned: 9.32 GB
    ```

    Now, you will compare the performance of running queries on the *nonpartitioned* data in the *yellow* table and the *partitioned* data in the *creditcard* table.

13. To query the nonpartitioned data in the *yellow* table, run the following query in a new query tab:

    ```
    SELECT sum (total), paytype FROM yellow
      WHERE paytype = '1' GROUP BY paytype;
    ```

    The run time and data scanned values are similar to the following:

    ```
    Time in queue: 215 ms
    Run time: 8.081 sec
    Data scanned: 9.32 GB
    ```

14. To query the partitioned data in the *creditcard* table, run the following query in a new query tab:

    ```
    SELECT sum (total), paytype FROM creditcard
      WHERE paytype = '1' GROUP BY paytype;
    ```

    The run time and data scanned values are similar to the following:

    ```
    Time in queue: 204 ms
    Run time: 2.259 sec
    Data scanned: 73.22 MB
    ```

Analysis: Notice that the results of each query are the same, but the query on the partitioned data took significantly less time because it scanned less data. Remember that you are charged for the the amount of data scanned for each query. So, since the last query scans less data due to the use of paritions, your cost is reduced.

Task 3 summary

In this task, you partitioned the dataset where the *paytype* field contained a specific value. Then, you ran queries on nonpartitioned and partitioned versions of the data and compared the results.

Task 4: Using Athena views

You might have noticed that Athena can create views. Creating and using views with data can help to simplify analysis because you can hide some of the complexity of queries from users.

Athena supports running only one SQL statement at a time, but you can use views to combine data from various tables. You can also use views to optimize query performance by experimenting with different ways to retrieve data and then saving the best query as a view.

In this task, you will do the following:

- Create a view to calculate the total dollar value of taxi fares that were paid with a credit card.
- Create a view to calculate the total dollar value of fares that were paid with cash.
- Retrieve all records from each of these views.

- Create a new view that joins the data from these two views.
- Preview the results of the join.

15. To create a view for the total dollar value of fares that were paid with a *credit card*, run the following query:

```
CREATE VIEW cctrips AS
  SELECT "sum"("fare") "CreditCardFares"
  FROM yellow
  WHERE ("paytype"='1');
```

    The run time and data scanned values are similar to the following:

```
Time in queue: 55 ms
Run time: 408 ms
Data scanned: -
```

16. To create a view for the total dollar value of fares that were paid with *cash*, run the following query:

```
CREATE VIEW cashtrips AS
  SELECT "sum"("fare") "CashFares"
  FROM yellow
  WHERE ("paytype"='2');
```

    The run time and data scanned values are similar to the following:

```
Time in queue: 68 ms
Run time: 378 ms
Data scanned: -
```

    Notice that the **Views** section on the left now lists the two views that you created: *cctrips* and *cashtrips*.

17. To select all records from the *cctrips* view, run the following query:

```
Select * from cctrips;
```

    The run time and data scanned values are similar to the following:

```
Time in queue: 158 ms
Run time: 9.071 sec
Data scanned: 9.32 GB
```

18. To select all records from the *cashtrips* view, run the following query:

```
Select * from cashtrips;
```

The run time and data scanned values are similar to the following:

```
Time in queue: 212 ms
Run time: 5.157 sec
Data scanned: 9.32 GB
```

Now you will learn how to create a view that joins data from two different views. You will use this new view to find out the total revenue from credit card payments compared to cash payments for two vendors.

19. To create a new view that joins the data, run the following query:

```
CREATE VIEW comparepay AS
WITH
  cc AS
    (SELECT sum(fare) AS cctotal,
        vendor
    FROM yellow
    WHERE paytype = '1'
    GROUP BY paytype, vendor),
    cs AS
    (SELECT sum(fare) AS cashtotal,
            vendor, paytype
    FROM yellow
    WHERE paytype = '2'
    GROUP BY paytype, vendor)

SELECT cc.cctotal, cs.cashtotal
FROM cc
JOIN cs
ON cc.vendor = cs.vendor;
```

The run time and data scanned values are similar to the following:

```
Time in queue: 54 ms
Run time: 435 ms
Data scanned: -
```

20. Preview the results from the join.
    - In the **Views** section on the left, to the right of the **comparepay** view, choose the ellipsis icon, and then choose **Preview View**.
    - If prompted about opening the query, choose **Open query**.

      The results display and are similar to the following:

      ```
      #   cctotal cashtotal
      1   584502884   250849783
      2   460097126   199181978
      ```

Task 4 summary

In this task, you learned how to create views in Athena. You created two views to calculate the total revenue of taxi fares that were paid with a credit card and paid with cash. You also learned how to use a view to join data from two other views. The skills that you learned in this task will help simplify analysis of data by using Athena.

Task 5: Creating Athena named queries by using CloudFormation

The data science team wants to share the queries that they built by using the taxi dataset. The team would like to share with other departments, but those departments use other accounts in AWS. Other departments have less experience with AWS, so the data science team would like to simplify the process of using Athena for other departments to use.

✎ **Note:** In this lab, you will focus on creating a CloudFormation template to create a named query in Athena. The template will not create the AWS Glue database or the tables and data within it.

You want to simplify the process to deploy queries. After some research, you determine that the best solution is to build a CloudFormation template for each query. The template can be shared with other departments and deployed as needed.

First, you want to experiment by building an example query and testing it by using Mary's IAM user. If you are successful, you will share the template with other departments.

21. Review and run the example query.

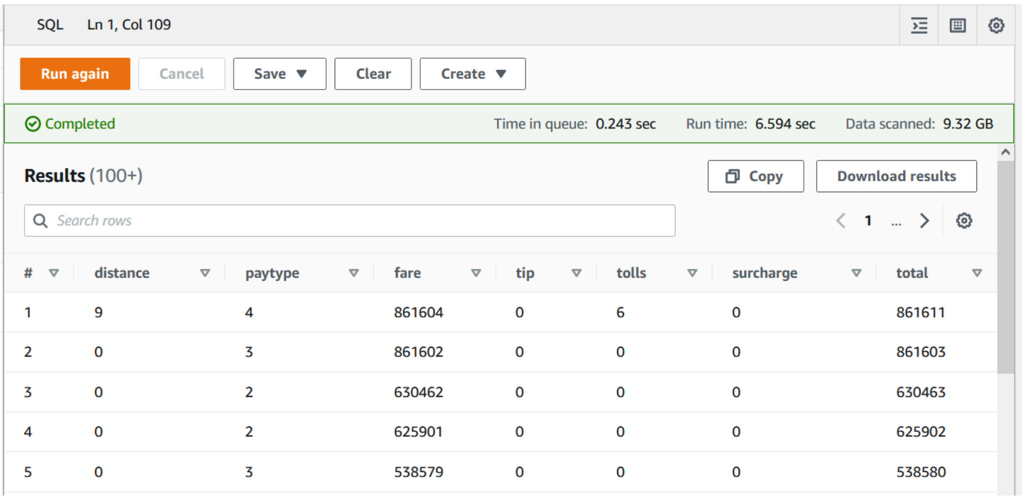
    Mary provides you with the following example query:

    ```
    SELECT distance, paytype, fare, tip, tolls, surcharge, total
    FROM yellow WHERE total >= 100.0 ORDER BY total DESC
    ```

    This query uses the *yellow* table of the *taxidata* database. The query does the following actions:

    - Selects all records in the *yellow* table where the total charge is greater than or equal to $100.00
    - Displays the records in descending order by the *total* field (so the most expensive trip is displayed first in the results)
    - Displays the following fields for the resulting records:
        - *distance*
        - *paytype*
        - *fare*
        - *tip*
        - *tolls*
        - *surcharge*
        - *total*
    - Run this example query in the Athena query editor.

      The results are similar to the following image:



Completed Time in queue: 0.243 sec Run time: 6.594 sec Data scanned: 9.32 GB

Results (100+)

#	distance	paytype	fare	tip	tolls	surcharge	total
1	9	4	861604	0	6	0	861611
2	0	3	861602	0	0	0	861603
3	0	2	630462	0	0	0	630463
4	0	2	625901	0	0	0	625902
5	0	3	538579	0	0	0	538580

The query is working as designed. Now, focus on creating the CloudFormation template for the query so that it can be reused and shared.

22. Navigate to the AWS Cloud9 integrated development environment (IDE).

    - In the AWS Management Console, in the search box next to **Services**, search for and choose **Cloud9** to open the AWS Cloud9 console.

      AWS Cloud9 environments are listed.

    - For the environment named **Cloud9 Instance**, choose **Open IDE**.

      A new browser tab opens and displays the AWS Cloud9 IDE.

23. Create a new CloudFormation template.

    - In the AWS Cloud9 IDE, choose **File** > **New File**.
    - Save the empty file as `athenaquery.cf.yml` but keep it open.
    - Copy and paste the following code into the file, and save the file:

```
AWSTemplateFormatVersion: 2010-09-09
Resources:
  AthenaNamedQuery:
    Type: AWS::Athena::NamedQuery
    Properties:
      Database: "taxidata"
      Description: "A query that selects all fares over $100.00 (US)"
      Name: "FaresOver100DollarsUS"
      QueryString: >
                    SELECT distance, paytype, fare, tip, tolls, surcharge, total
                    FROM yellow
                    WHERE total >= 100.0
                    ORDER BY total DESC
Outputs:
  AthenaNamedQuery:
    Value: !Ref AthenaNamedQuery
```

Examine the command to see what is being created. The command creates a CloudFormation templated named *athenaquery.cf.yml*. This template will create a CloudFormation stack that does the following:

- Creates an Athena query

- Uses a pre-existing database in Athena named *taxidata* (**Note:** For another user to use this template, the *taxidata* database must already exist in their AWS account.)
- Names the query *FaresOver100DollarsUS*
- Uses the example query that you tested previously in this task
- Outputs the details of the query to the **Outputs** tab of the stack once the stack is created

24. To validate the CloudFormation template, run the following command in the AWS Cloud9 terminal:

```
aws cloudformation validate-template --template-body
file://athenaquery.cf.yml
```

✎ **Note:** If you receive an error that says *YAML not well-formed*, check the value of the name of the query and also the tabs and spacing for each line. YAML documents require exact spacing, and the parser will encounter errors if the spacing doesn't match.

If the template is validated, the following output displays:

```
{
    "Parameters": []
}
```

ℹ **Important:** Don't go to the next step until the template is validated.

Now you will use the template to create a CloudFormation stack. A *stack* implements and manages the group of resources that are outlined in a template. With a stack, you can manage the state and dependencies of those resources together. Think of a CloudFormation template as a blueprint. Then, the stack is the actual instance of the template that is registered in AWS and actually creates the resources.

27. To create the stack, run the following command:

```
aws cloudformation create-stack --stack-name athenaquery  --
template-body file://athenaquery.cf.yml
```

If the stack is validated, the CloudFormation Amazon Resource Name (ARN) displays in the output, similar to the following:

```
{
    "StackId": "arn:aws:cloudformation:us-east-1:338778555682:stack/athenaquery/2d8cec90-5c42-11ec-8fbf-12034b0079a5"
}
```

The CloudFormation create-stack command creates the stack and deploys it. If validation passes and nothing causes the stack creation to roll back, proceed to the next step.

💡 **Tip:** To check the progress of stack creation, navigate to the CloudFormation console. In the navigation pane, choose **Stacks**, and look for the status of the *athenaquery* stack.

28. Confirm the named query resource that the CloudFormation stack created.

    - To verify that the named query was created, run the following command in the AWS Cloud9 terminal.

      ```
      aws athena list-named-queries
      ```

      The output is similar to the following:

      ```
      {
          "NamedQueryIds": [
            "644f6c10-bf57-48a2-a0ec-a3de179db511"
          ]
      }
      ```

      The named query ID is a unique identifier in AWS for the query that you created by using the CloudFormation stack.

    - Copy the query ID to a text editor.

    - To retrieve the details of the named query, including the SQL statement that is associated with it, run the following command. Replace <*QUERY-ID*> with the ID that you saved to a text editor.

      ```
      aws athena get-named-query --named-query-id <QUERY-ID>
      ```

      The output is similar to the following:

```
{
"NamedQuery": {
    "Name": "FaresOver100DollarsUS",
    "Description": "A query that selects all fares over $100.00 (US)",
    "Database": "taxidata",
    "QueryString": "SELECT distance, paytype, fare, tip, tolls, surcharge, total FROM yellow WHERE total >= 100.0 ORDER BY total DESC",
    "NamedQueryId": "644f6c10-bf57-48a2-a0ec-a3de179db511",
    "WorkGroup": "primary"
}
}
```

- To save the named query ID as a bash variable, run the following command. Replace <*QUERY-ID*> with the named query ID that you copied earlier. This will make it easier to use the ID in commands in later steps:

```
NQ=<QUERY-ID>
```

- Confirm the named query id was stored as the bash variable **NQ**. Run the command below.

```
echo $NQ
```

  There result is similar to the following:

```
644f6c10-bf57-48a2-a0ec-a3de179db511
```

Task 5 summary

In this task, you learned how to integrate an Athena named query into a CloudFormation template. You also learned how to validate and deploy a template to create a named query in a CloudFormation stack.

Many companies use multiple accounts with AWS to maintain separate development, testing, and production environments. Isolating these environments helps to ensure that teams follow best practices. Building queries in a development account and then testing them in a controlled account with production data can help to ensure that the query is designed as intended and generates the necessary insights. After validating the query, you can use DevOps best practices with CloudFormation to quickly move it to production so that the appropriate business stakeholders can reuse it without having to build from the beginning.

Task 6: Reviewing the IAM policy for Athena and AWS Glue access

Now that you have created the Athena named query by using CloudFormation, review the IAM policy for the query to ensure that others can use it in production.

Note: The IAM policy for the query was created for you; you don't have the ability to create IAM policies in the lab environment.

29. Review the *Policy-For-Data-Scientists* policy in IAM.
    - In the search box to the right of **Services**, search for and choose **IAM** to open the IAM console.
    - In the navigation pane, choose **Users**.

      Notice that *mary* is one of the IAM users that is listed. This user is part of the *DataScienceGroup* IAM group.
    - Choose the user name link for **mary**.
    - On the **Permissions** tab, choose the link for the **Policy-For-Data-Scientists** policy.

      The *Policy-For-Data-Scientists* details page opens. Review the permissions that are associated with this policy. Notice that the permissions provide limited access for only the Athena, CloudFormation, Cloud9, AWS Glue, and Amazon S3 services.

Task 6 summary

In this task, you reviewed the IAM policy for the *DataScienceGroup* IAM group.

The policy contains permissions for limited access to Amazon S3, AWS Glue, and Athena. The policy is attached to the IAM group that the *mary* IAM user is associated with. The policy allows her to list named queries. The policy also includes the following permissions:

- **For Amazon S3:** Access buckets, list bucket contents, and read and write objects, but not create a bucket
- **For AWS Glue:** Create databases, and tables within the databases
- **For CloudShell:** use CloudShell, for example issue AWS CLI commands
- **For CloudFormation:** Create stacks and validate templates
- **For Athena:** Create a Data Catalog, create and get named queries, and start queries

The policy could be used as an example policy for users who intend to create and use Athena queries and views with a dataset by loading it into an AWS Glue database. As with all services in AWS, IAM users must have the appropriate permissions applied to be able to perform actions.

Task 7: Confirming that Mary can access and use the named query

Now that you have reviewed the IAM policy, you will use it to test another user's access to the named query and their ability to use the query in the AWS CLI.

30. Retrieve the credentials for the *mary* IAM user.

    - In the search box next to ▦ **Services**, search for and choose **CloudFormation**.
    - In the navigation pane, choose **Stacks**.
    - Choose the link for the stack that created the lab environment. The stack name includes a random string of letters and numbers, and the stack should have the oldest creation time.
    - On the stack details page, choose the **Outputs** tab.

      ✎ **Note:** When you create a CloudFormation template, you can choose to output information about the resources that the template will create. The CloudFormation template that created the resources in your lab environment output the access key and secret access key for the *mary* user.

    - Copy the value of **MarysAccessKey** to your clipboard.
    - Return to the AWS Cloud9 terminal.
    - To create a variable for the access key, run the following command. Replace <*ACCESS-KEY*> with the value from your clipboard.

      ```
      AK=<ACCESS-KEY>
      ```

- Return to the CloudFormation console, and copy the value of **MarysSecretAccessKey** to your clipboard.
- Return to the AWS Cloud9 terminal.
- To create a variable for the secret access key, run the following command. Replace <*SECRET-ACCESS-KEY*> with the value from your clipboard.

  ```
  SAK=<SECRET-ACCESS-KEY>
  ```

  To test whether the *mary* user can perform the get-named-query command, you can pass the user's credentials as bash variables (*AK* and *SAK*) with the command. You will also use a bash variable (*NQ*) to include the named query ID. The API will then try to perform that command as the specified user.

31. To test whether Mary can use the named query, run the following command:

    ```
    AWS_ACCESS_KEY_ID=$AK AWS_SECRET_ACCESS_KEY=$SAK aws athena get-named-query --named-query-id $NQ
    ```

    The output is similar to the following and looks like the output that was displayed after you ran the command earlier:

    ```
    {
        "NamedQuery": {
            "Name": "FaresOver100DollarsUS",
            "Description": "A query that selects all fares over $100.00 (US)",
            "Database": "taxidata",
            "QueryString": "SELECT distance, paytype, fare, tip, tolls, surcharge, total FROM yellow WHERE total >= 100.0 ORDER BY total DESC",
            "NamedQueryId": "644f6c10-bf57-48a2-a0ec-a3de179db511",
            "WorkGroup": "primary"
        }
    }
    ```

    This result confirms that Mary has access to the named query that you created and deployed by using CloudFormation.

Task 7 summary

Congratulations! You learned how to create an Athena named query by using CloudFormation and to deploy it to users with a secure IAM policy. Because the query is in a CloudFormation template, you can reuse the template to create and deploy the query in any AWS account and change the parameters as desired.

Update from the team

The team is happy with what you have learned and demonstrated by using Athena, AWS Glue, and CloudFormation.

Submitting your work

32. To record your progress, choose **Submit** at the top of these instructions.
33. When prompted, choose **Yes**.

    After a couple of minutes, the grades panel appears and shows you how many points you earned for each task. If the results don't display after a couple of minutes, choose **Grades** at the top of these instructions.

    ⚠ **Important:** Some of the checks made by the submission process in this lab will only give you credit if it has been at least 5 minutes since you completed the action. If you do not receive credit the first time you submit, you may need to wait a couple minutes and the submit again to receive credit for these items.

    💡 **Tip:** You can submit your work multiple times. After you change your work, choose **Submit** again. Your last submission is recorded for this lab.
34. To find detailed feedback about your work, choose **Submission Report**.

Lab complete

Congratulations! You have completed the lab.

35. At the top of this page, choose ■ **End Lab**, and then choose Yes to confirm that you want to end the lab.

A message panel indicates that the lab is terminating.

36. To close the panel, choose **Close** in the upper-right corner.

Additional resources

For more information about the services and concepts covered in this lab, see the following resources:

- Amazon Athena Documentation
- Getting Started with AWS Glue
- Presto Documentation
- DML Queries, Functions, and Operators
- Amazon Athena Pricing
- Using a SerDe
- Columnar Storage Formats
- Bucketing vs. Partitioning
- Working with Views in Athena
- Policies and Permissions in IAM