Tanawin Siriwan st123975 MMI
GitHub : https://github.com/TwoTanawin/mlops-lab5.git

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.

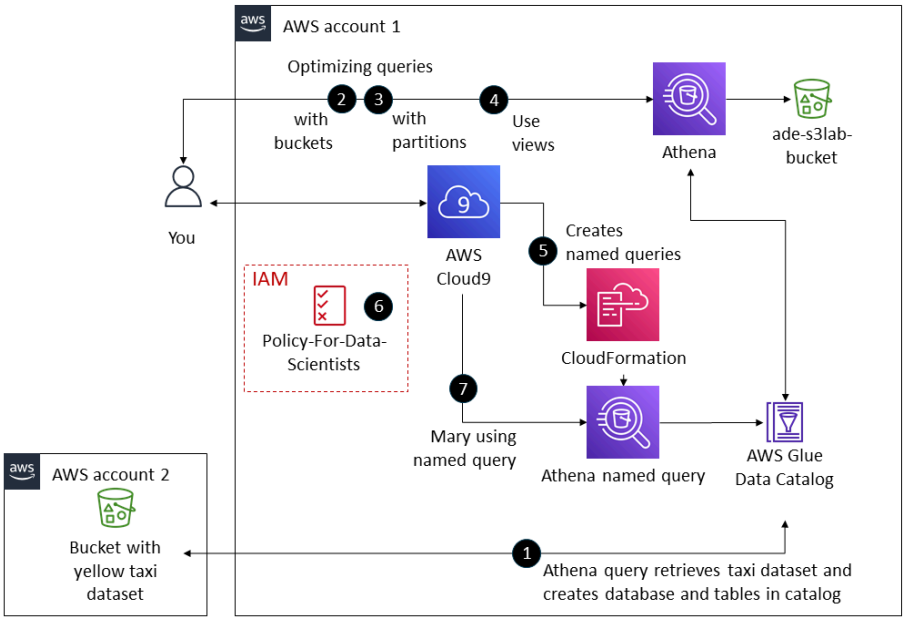
After completing this lab, you should be able to do the following:
- Use the Athena query editor to create an AWS Glue database and table.
- Define the schema for the AWS Glue database and associated tables by using the Athena bulk add columns feature.
- Configure Athena to use a dataset that is located 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.

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. You will use the data for your POC.

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.



Note: currently, AWS Cloud9 is not available in AWS services so we can use other IDE, for example, VScode to fix or rewrite your program.

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.

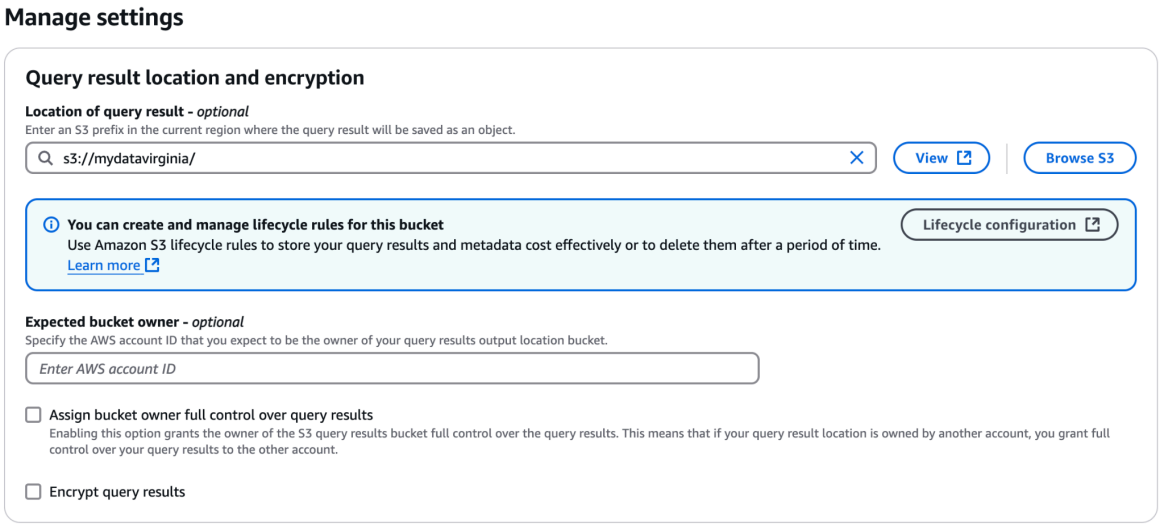
4. In the AWS Management Console, in the search box to the right of Services, search for and choose Athena to open the Athena console.

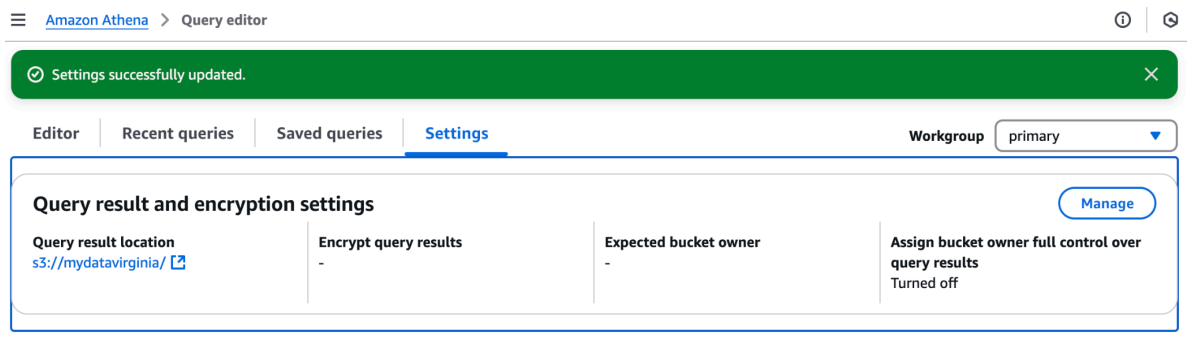
5. Specify an S3 bucket for query results.

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**.



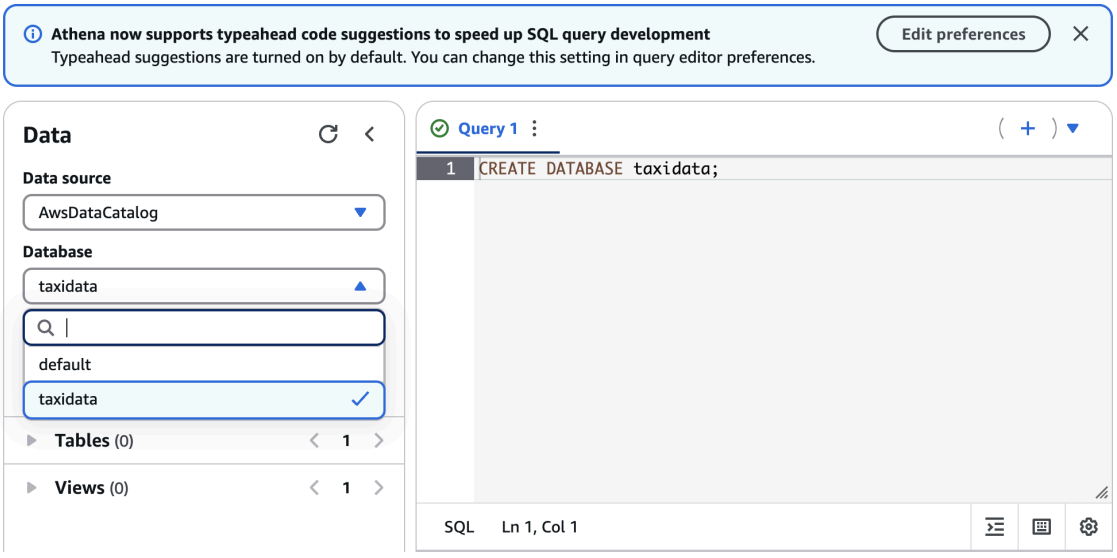


6. Create an AWS Glue database by using the Athena query editor.

- Choose the **Editor** tab.
- In the **Query 1** section, enter the following SQL command:

```
CREATE DATABASE taxidata;
```

- Choose Run.
- The message Query successfully 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:

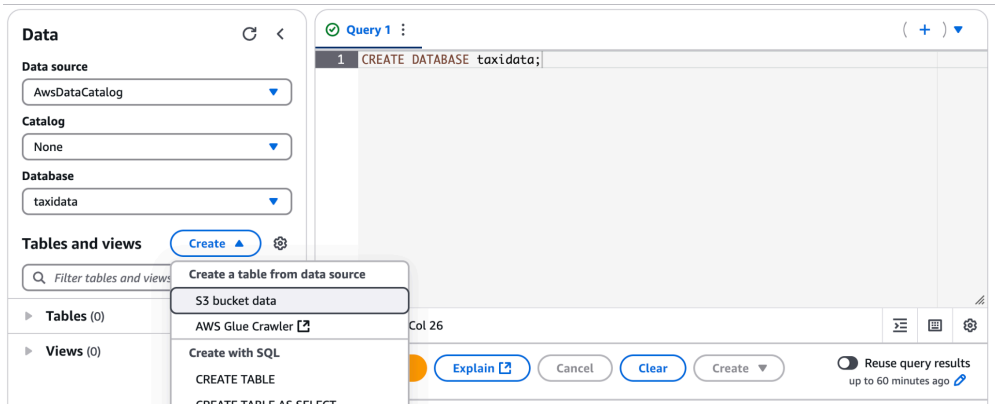
- **Table name:** Enter yellow
- **Description:** Enter Table for taxi data
- **Database configuration:** Select **Choose an existing database**, and then choose **taxidata** from the dropdown list.
- **Location of input dataset:** Copy the following link into the field:

  s3://datafortable
  Note: s3 location may be different.

- **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.



Create table from S3 bucket data Info

Table details

Table name

yellow

Table name must be from 1-128 characters and must be unique. Valid characters are a-z, A-Z, 0-9, _(underscore). Table names tend to correspond to the directory where the data will be stored.

Description - *optional*

Table for taxi data

Table description must be from 1-1024 characters. 1005 characters remaining.

Database configuration Info

Choose an existing database or create a new database

Choose to access an existing database or to create a new database in order to create a new table. Athena stores the table schema in the AWS Glue Data Catalog.

- ○ Create a database
- ◉ Choose an existing database

taxidata

Dataset Info

Location of input data set

s3://mydatavirginia

View | Browse S3

Input the path to the data set you want to process on Amazon S3. For example if your data is stored at s3://input-data-set/logs/1.csv, please enter s3://input-data-set/logs/. If your data is already partitioned, e.g. s3://input-data-set/logs/year=2004/month=12/day=11/ just input the base path s3://input-data-set/logs/

Data format Info

Table type

Apache Hive

File format

CSV

SerDe library

org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe

SerDe properties - *optional*

Name	Value	
field.delim	,	Remove

Add SerDe property

Column details

Column name must be from 1-128 characters. Valid characters are a-z, A-Z, 0-9, _(underscore). Certain advanced column types (namely, structs) are not exposed in this interface.

Column name	Column type	Description - *optional*	
vendor	string	*Enter description*	Remove
pickup	timestamp	*Enter description*	Remove
dropoff	timestamp	*Enter description*	Remove
count	int	*Enter description*	Remove
distance	int	*Enter description*	Remove
ratecode	string	*Enter description*	Remove
storeflag	string	*Enter description*	Remove
pulocid	string	*Enter description*	Remove
dolocid	string	*Enter description*	Remove
paytype	string	*Enter description*	Remove
fare	decimal	*Enter description*	Remove

Precision - *optional*: 0

Scale - *optional*:

Column name	Column type	Description - *optional*		Precision - *optional*	Scale - *optional*
extra	decimal	Enter description	Remove	0	
mta_tax	decimal	Enter description	Remove	0	
tip	decimal	Enter description	Remove	0	
tolls	decimal	Enter description	Remove	0	
surcharge	decimal	Enter description	Remove	0	
total	decimal	Enter description	Remove	0	

Add a column | Bulk add columns

▶ **Table properties - *optional*** Info

▶ **Partition details - *optional*** Info

Partitions are a way to group specific information together. Column name must be from 1-128 characters. Valid characters are a-z, A-Z, 0-9, _(underscore). Certain advanced column types (namely, structs) are not exposed in this interface.

▶ **Bucketing - *optional***

Preview table query

The preview table query will be populated in the query editor. Creating tables allows you to be ready for real-time querying in the query editor.

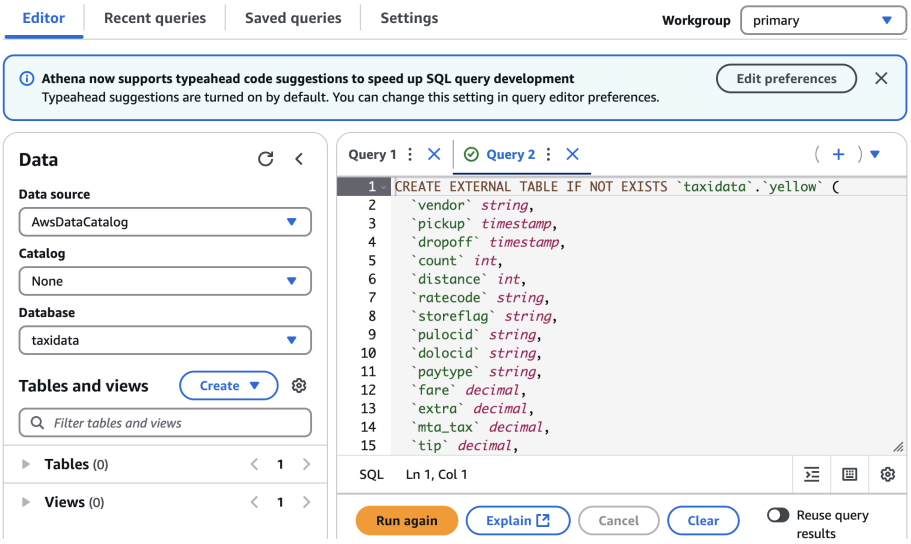
```
  `storeflag` string,
  `pulocid` string,
  `dolocid` string,
  `paytype` string,
  `fare` decimal,
  `extra` decimal,
  `mta_tax` decimal,
  `tip` decimal,
  `tolls` decimal,
  `surcharge` decimal,
  `total` decimal
) COMMENT "Table for taxi data"
```

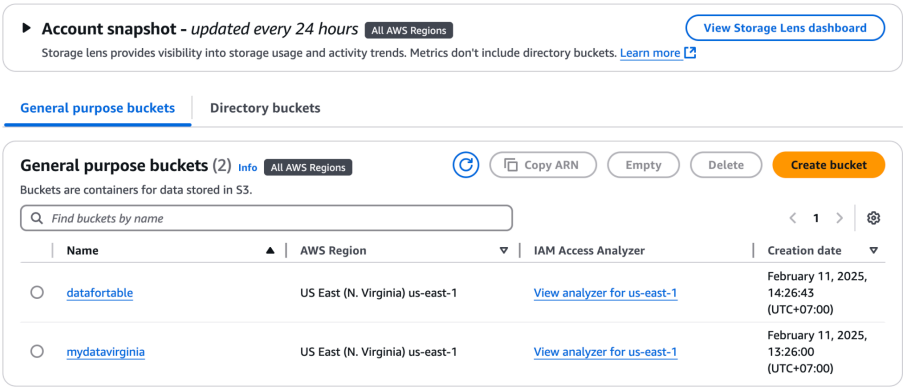
- In the Preview table query section, view the preview table query text, which matches the following text:

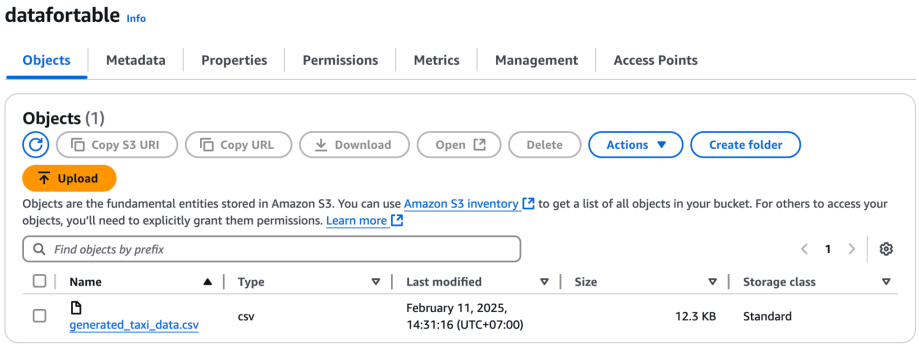
```
CREATE EXTERNAL TABLE IF NOT EXISTS `taxidata`.`yellow` (
  `vendor` string,
  `pickup` timestamp,
  `dropoff` timestamp,
  `count` int,
  `distance` int,
  `ratecode` string,
  `storeflag` string,
  `pulocid` string,
  `dolocid` string,
  `paytype` string,
  `fare` decimal,
  `extra` decimal,
  `mta_tax` decimal,
  `tip` decimal,
  `tolls` decimal,
  `surcharge` decimal,
  `total` decimal
) COMMENT "Table for taxi data"
ROW FORMAT SERDE
'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'
WITH SERDEPROPERTIES ('field.delim' = ',')
STORED AS INPUTFORMAT
'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION 's3://datafortable/'
TBLPROPERTIES ('classification' = 'csv');
```

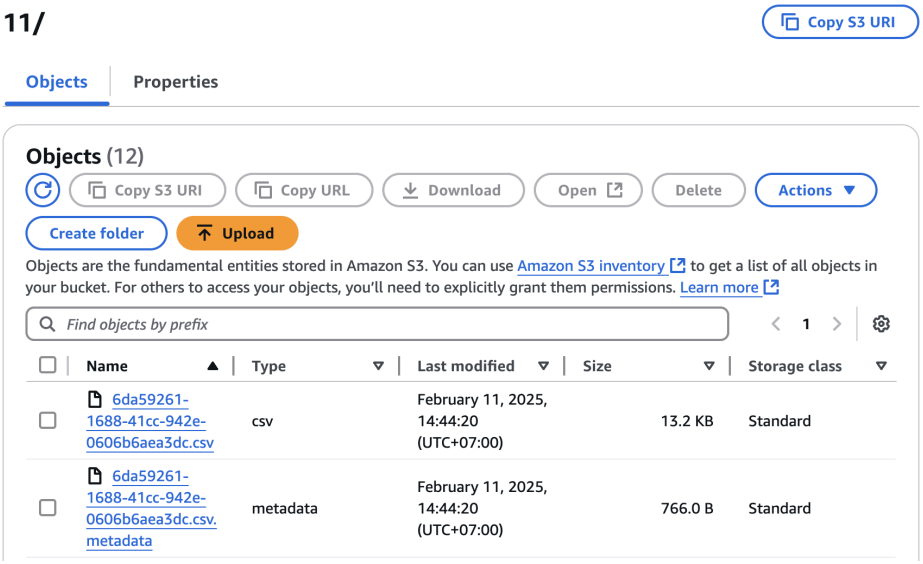
Note that the data is not encrypted. Serde is a data serialization format, and the SERDE property specifies that the data must be comma delimited, which it is.

- 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

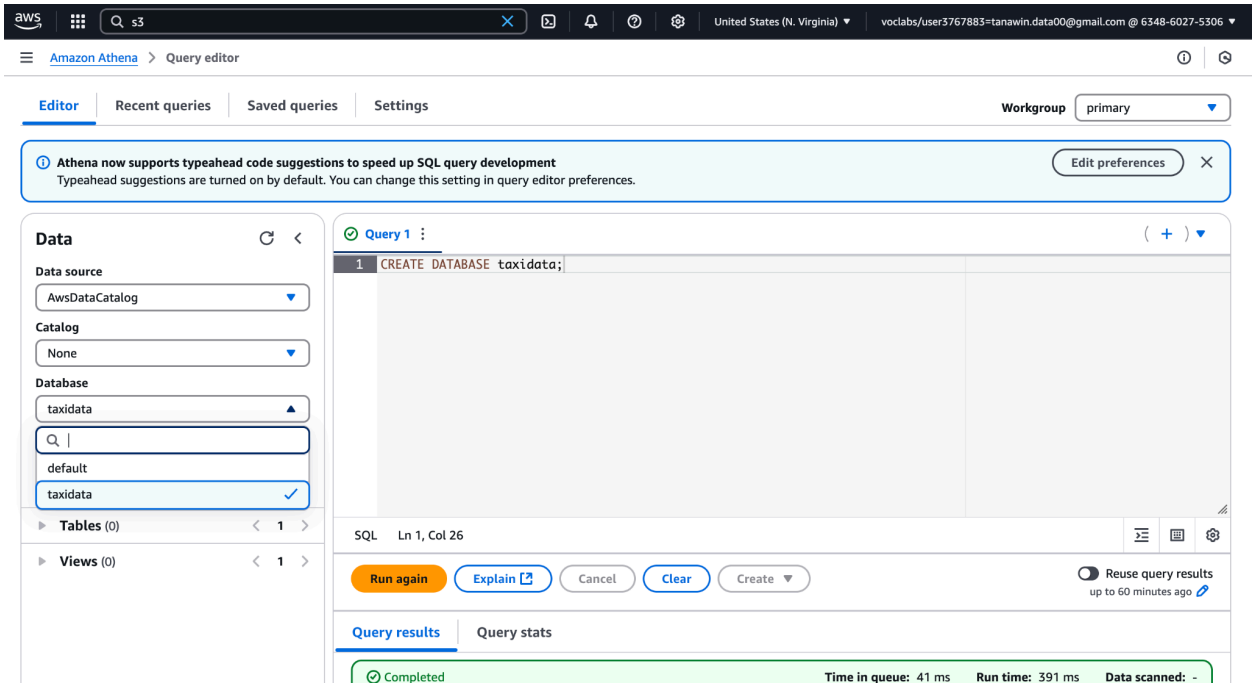
In this task, you created an AWS Glue database and table by using the Athena query editor. You connected an Amazon S3 dataset to the table and defined the schema for the table by using the bulk add columns feature. After you created the table, you learned how to preview the data by using the preview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

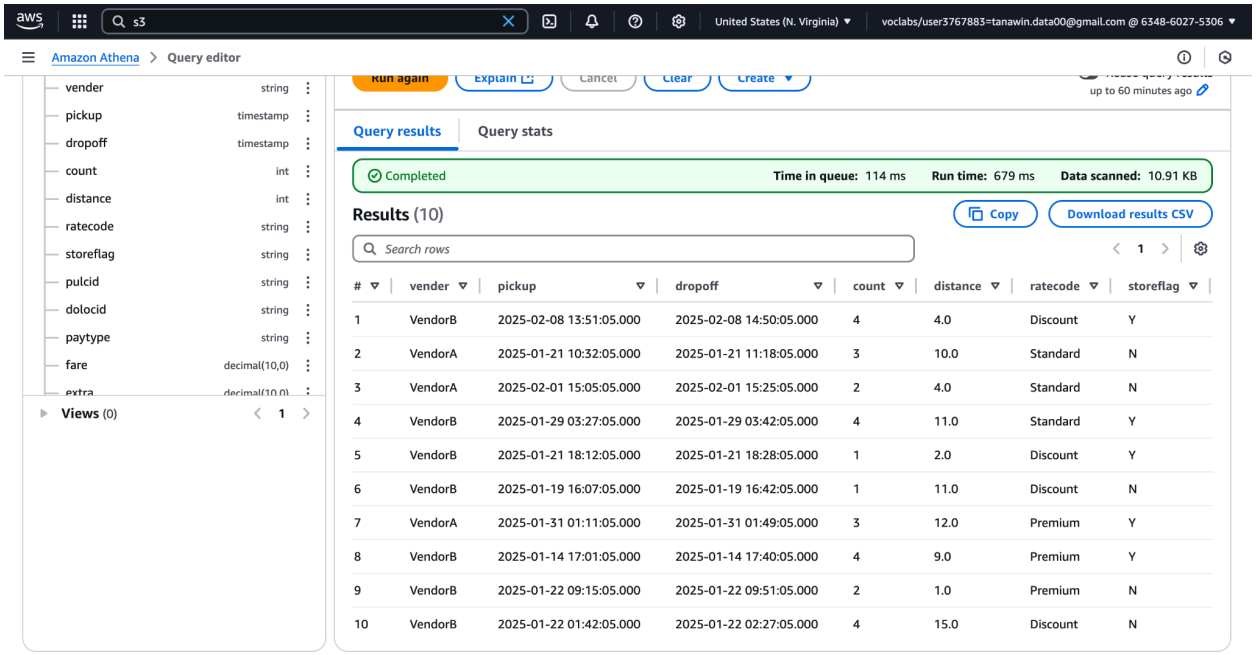
Capture your result

- **First 10 rows of the read Athena query from yellow.**

This figure shows the created taxi database.



This figure shows queried data from the dataset.



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).
- **Cardinality in SQL** refers to either the **"number of unique values"** in a column of a table or the **"size of the relationship between two tables"** in a relational database.
    - **High Cardinality:** A column with many unique values, such as **ID numbers, phone numbers, or email addresses**.
    - **Low Cardinality:** A column with many repeated values, such as **gender (Male/Female) or marital status (Married/Single)**.
- 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 2025 and how long it takes to query the entire dataset for the January 2025 data.

9. Create a table for the January 2025 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 TABLE jan AS
SELECT *
FROM yellow WHERE pickup
BETWEEN TIMESTAMP '2025-01-01 00:00:00'
  AND TIMESTAMP '2025-02-01 00:00:01'
```

The message Query successful displays. Athena created a new table named jan, which contains the taxi data for only the month of January 2025. 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 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 '2025-01-01 00:00:00'
  and TIMESTAMP '2025-02-01 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

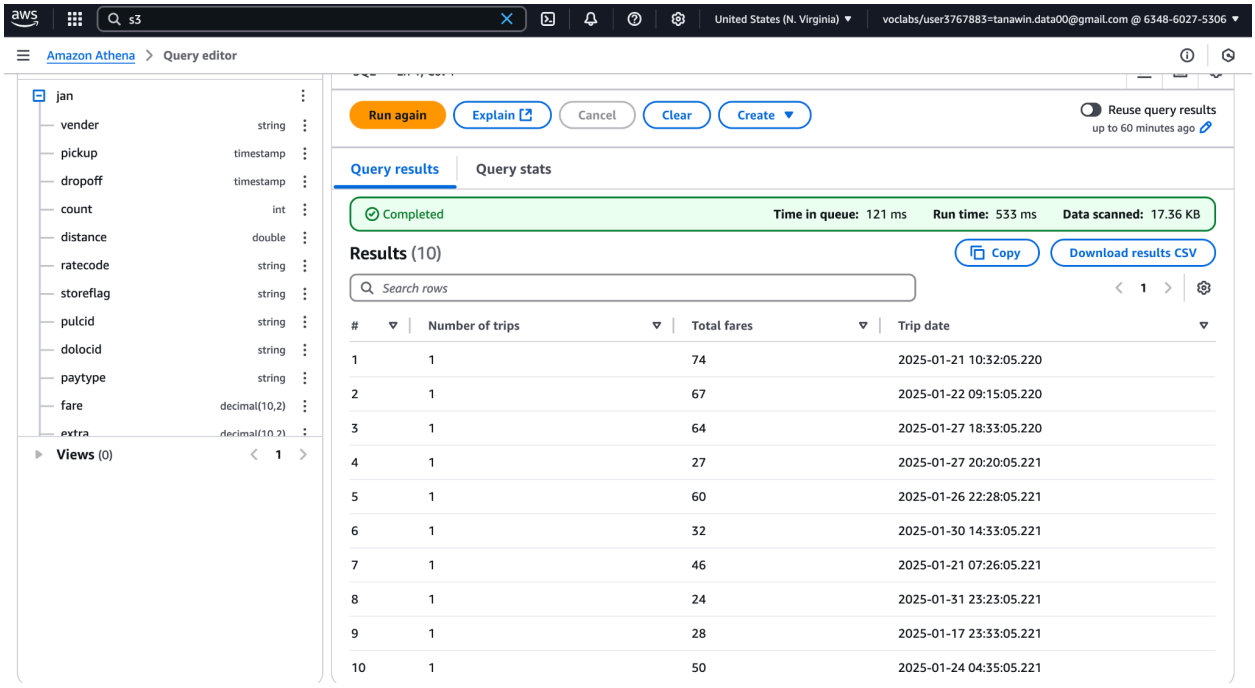
In this task, you created a table for the January data. You learned how to optimize queries by using various strategies, such as dividing data into multiple buckets and partitioning data.

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 data, which was divided into a separate bucket, and compared the results to the baseline.

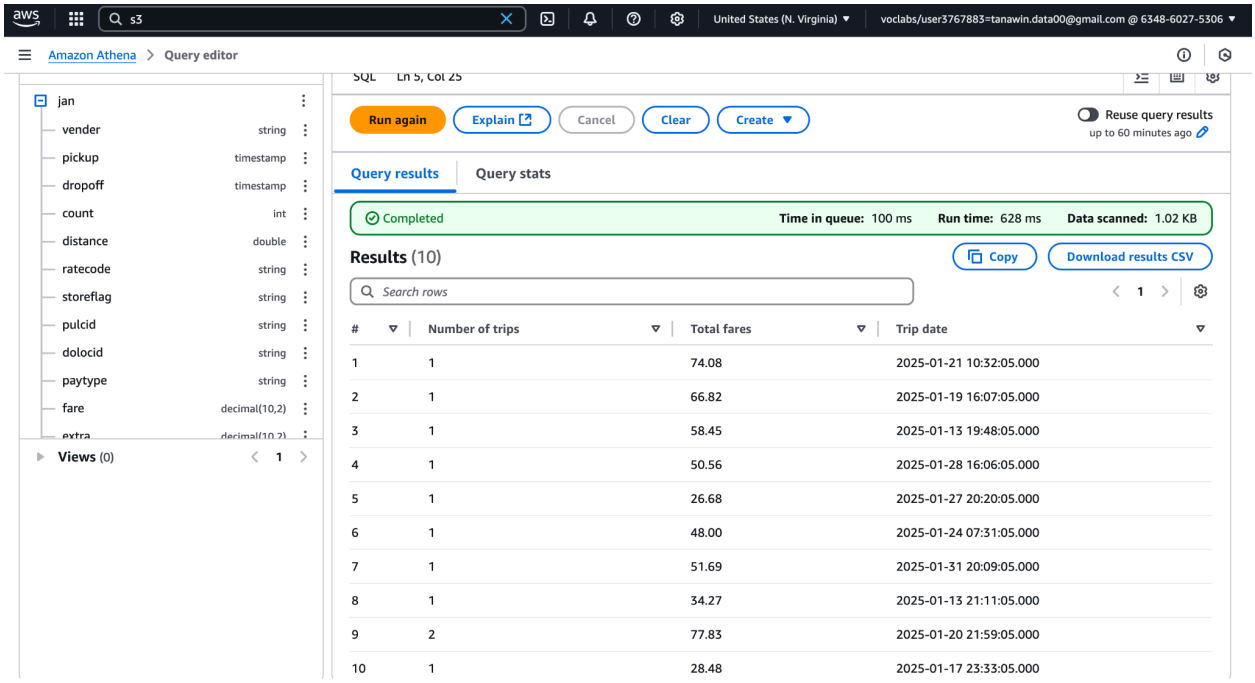
Capture your result

- **The result and time from no. 11 and no 12.**

This figure shows result from 11



This figure shows result from 12



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.

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 = 'Credit';
```

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 = 'Credit' GROUP BY paytype;
```

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

```
Time in queue: 70 ms
Run time: 497 ms
Data scanned: 12.25 KB
```

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 = 'Credit' GROUP BY paytype;
```

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

```
Time in queue: 69 ms
Run time: 418 sec
Data scanned: 0.20 KB
```

Time in queue
This refers to the amount of time a query has to wait in the queue before it starts running. Normally, if the system has other queries being processed or is under high load, Athena will queue your query.
Run time
This is the time taken to execute the query from start to finish.
Data scanned
This refers to the amount of data Athena needs to scan to process the query.
If the value is "-" (no data), it could mean:

- The query did not read actual data (e.g., commands like SHOW TABLES or DESCRIBE).
- No data matched the query's conditions.
- Partitions were used, and Athena did not need to scan all the data.
- The query might have been optimized to avoid reading actual files.

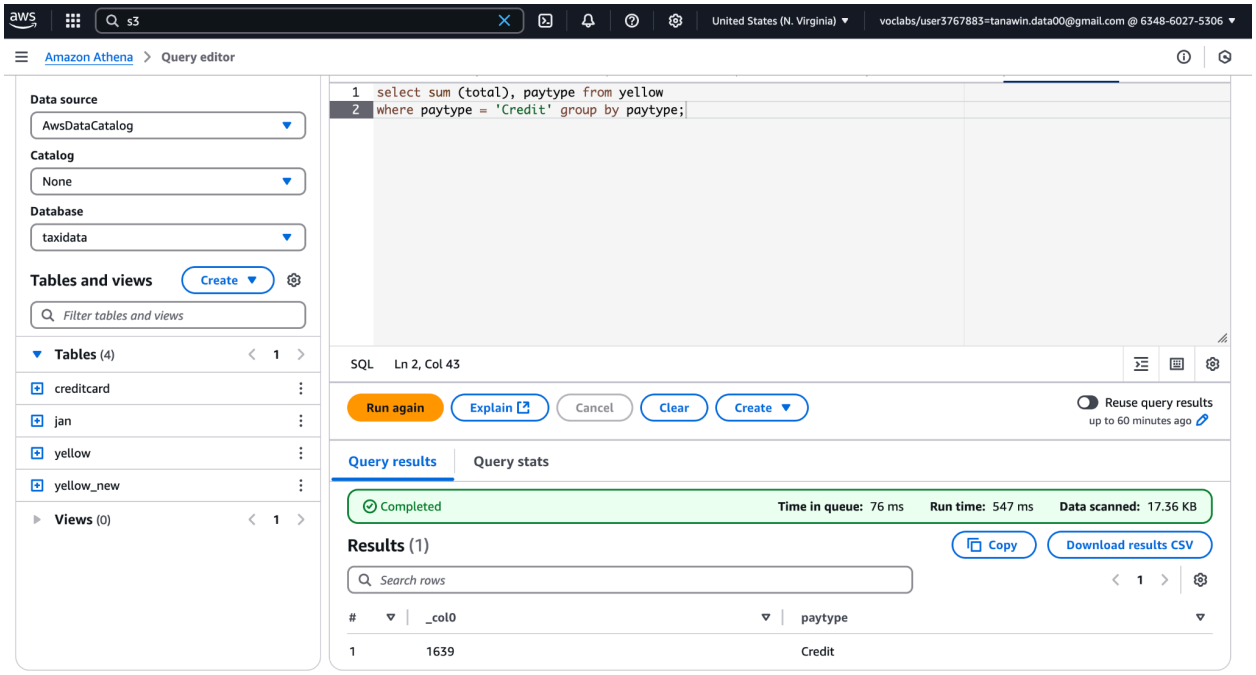
Analysis: Notice that the results of each query are the same, but the query on the partitioned data took significantly less time because it scanned less data. Remember that you are charged for the the amount of data scanned for each query. So, since the last query scans less data due to the use of paritions,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

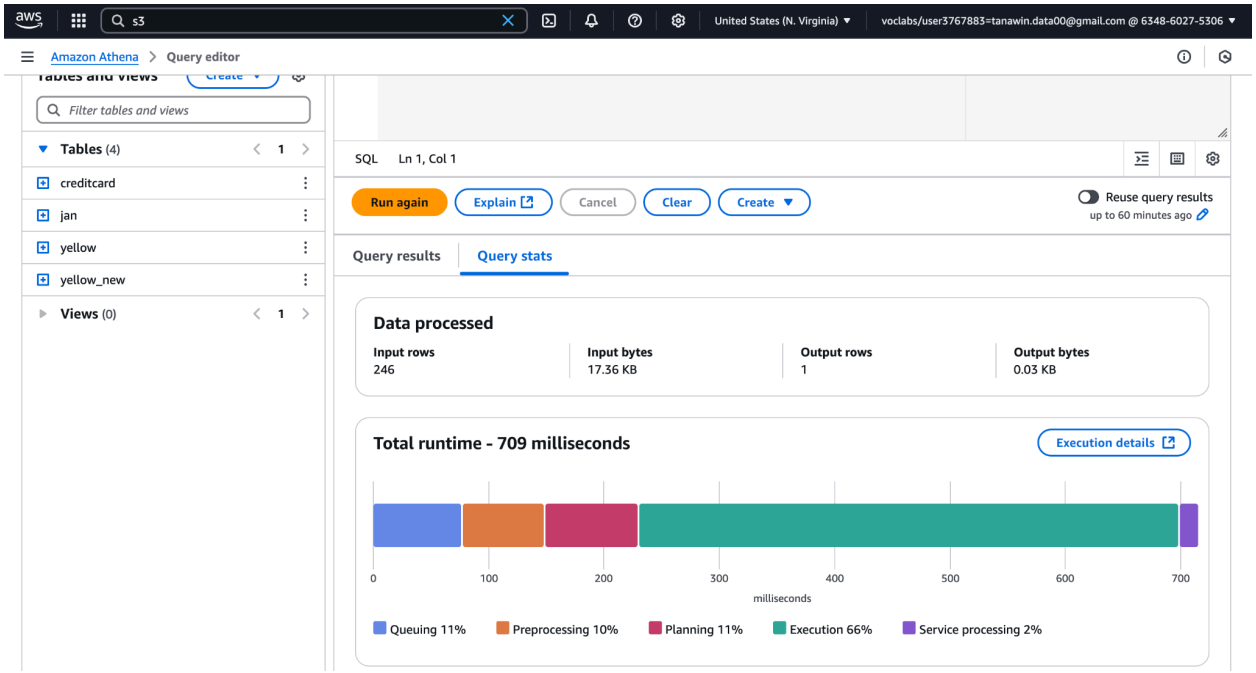
Capture your result

- **The result and time from no. 13 and no 14.**

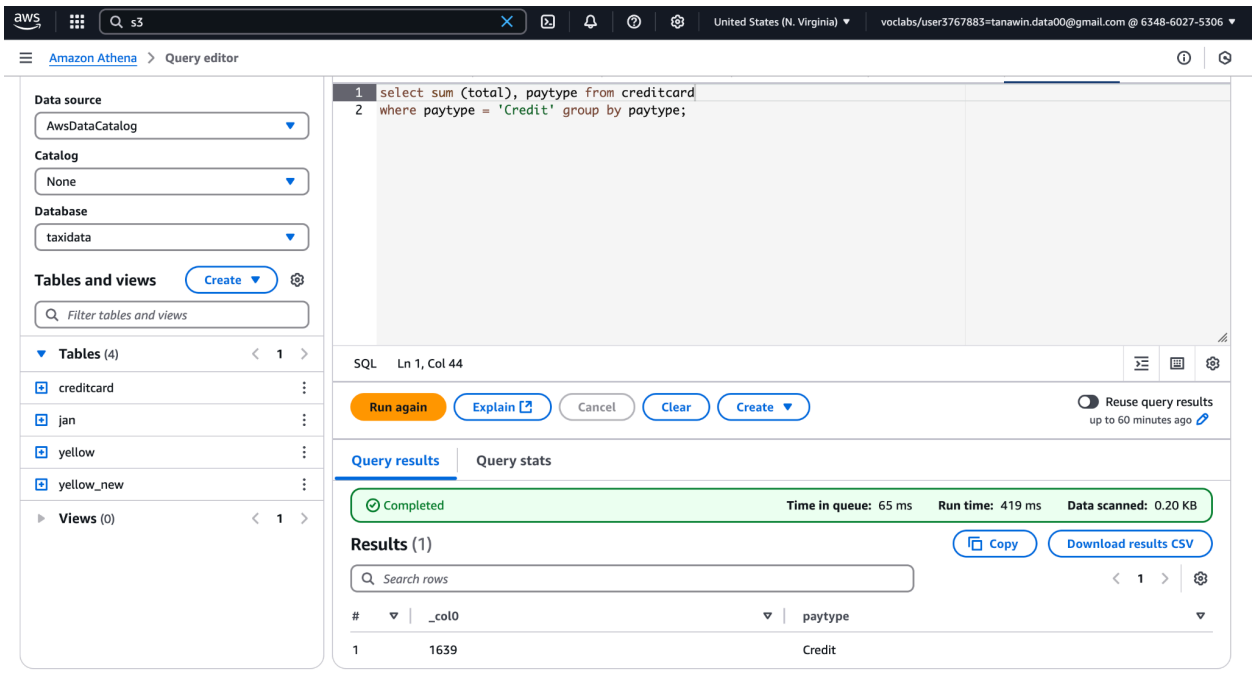
This figure shows result from 13



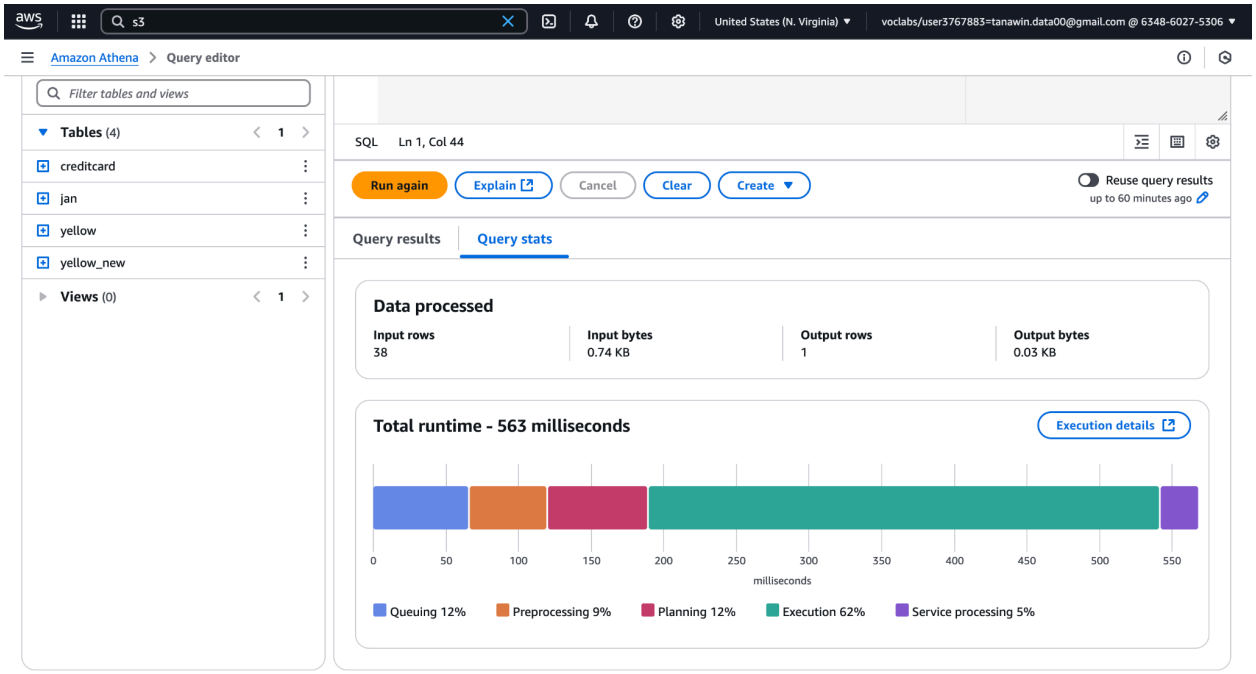
This figure shows results from 13 query performance.



This figure shows result from 14



This figure shows results from 14 query performance.



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"='Credit');
```

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

```
Time in queue: 83 ms
Run time: 513 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"='Digital');
```

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

```
Time in queue: 47 ms

Run time: 599 ms

Data scanned: -
```

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: 108 ms
Run time: 650 ms
Data scanned: 12.25 KB
```

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: 103 ms

Run time: 660 ms

Data scanned: 12.25 KB
```

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 = 'Credit'

   GROUP BY paytype, vendor),

   cs AS

   (SELECT sum(fare) AS cashtotal,

           vendor, paytype

   FROM yellow

   WHERE paytype = 'Digital'

   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: 36 ms
Run time: 547 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   496   612
2   441   346
```

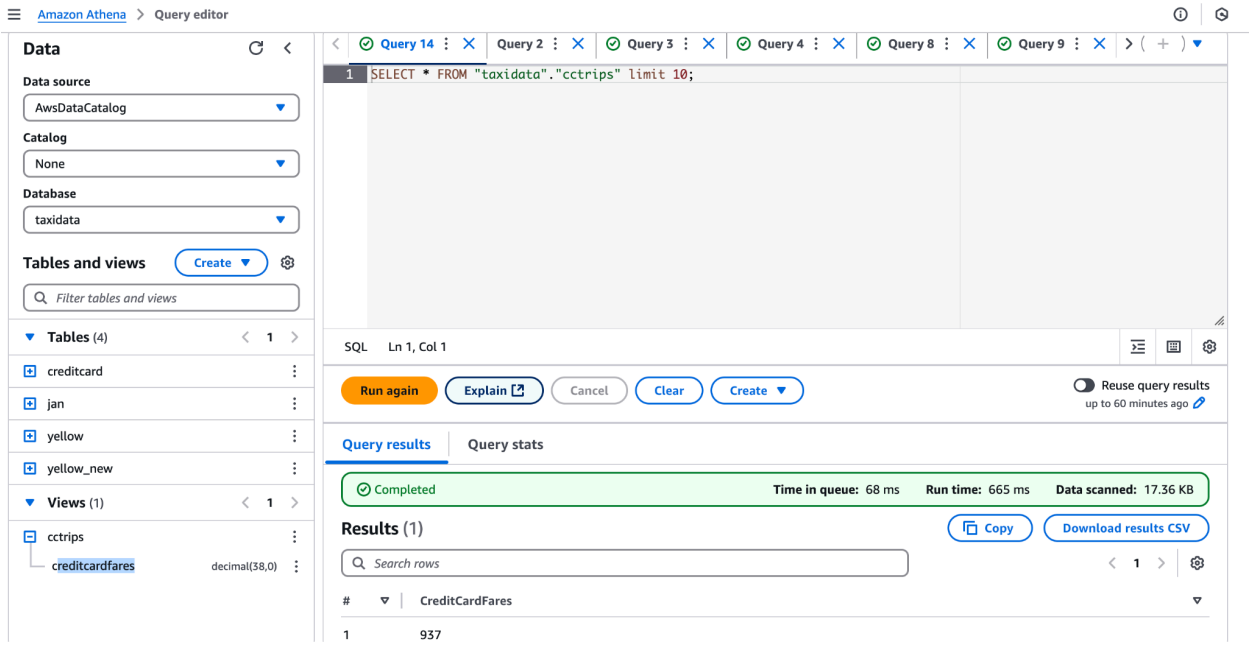
Task 4 summary

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.

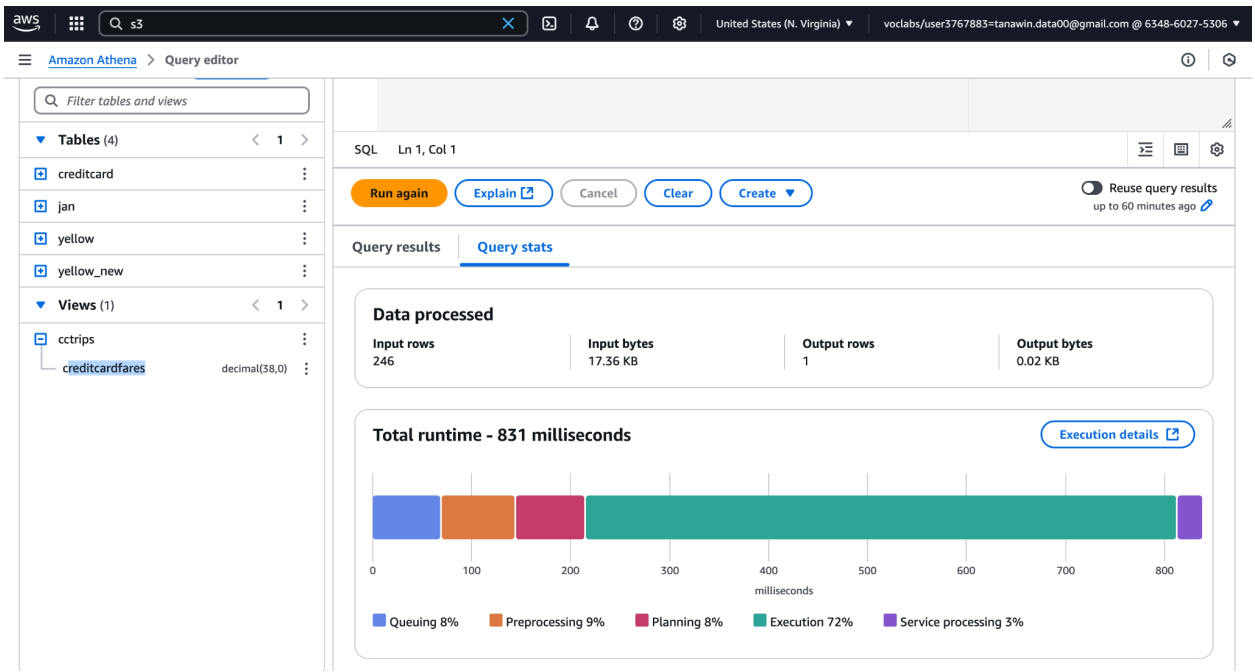
Capture your result

- **The result and time from no. 19.**

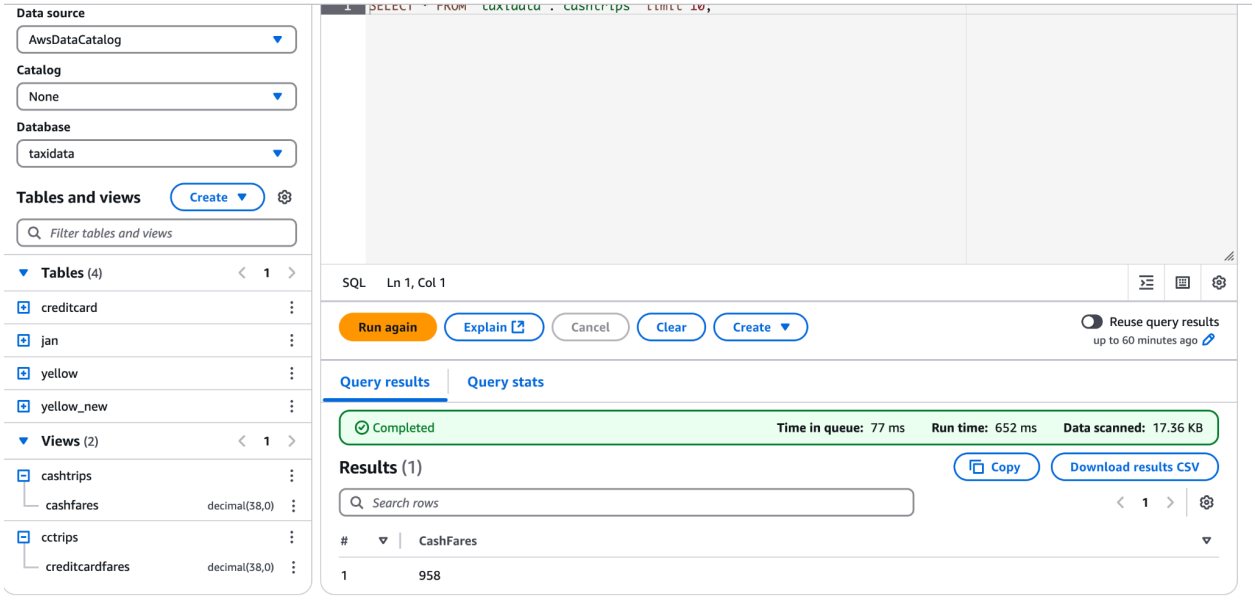
This figure shows results fetched from cctrips.



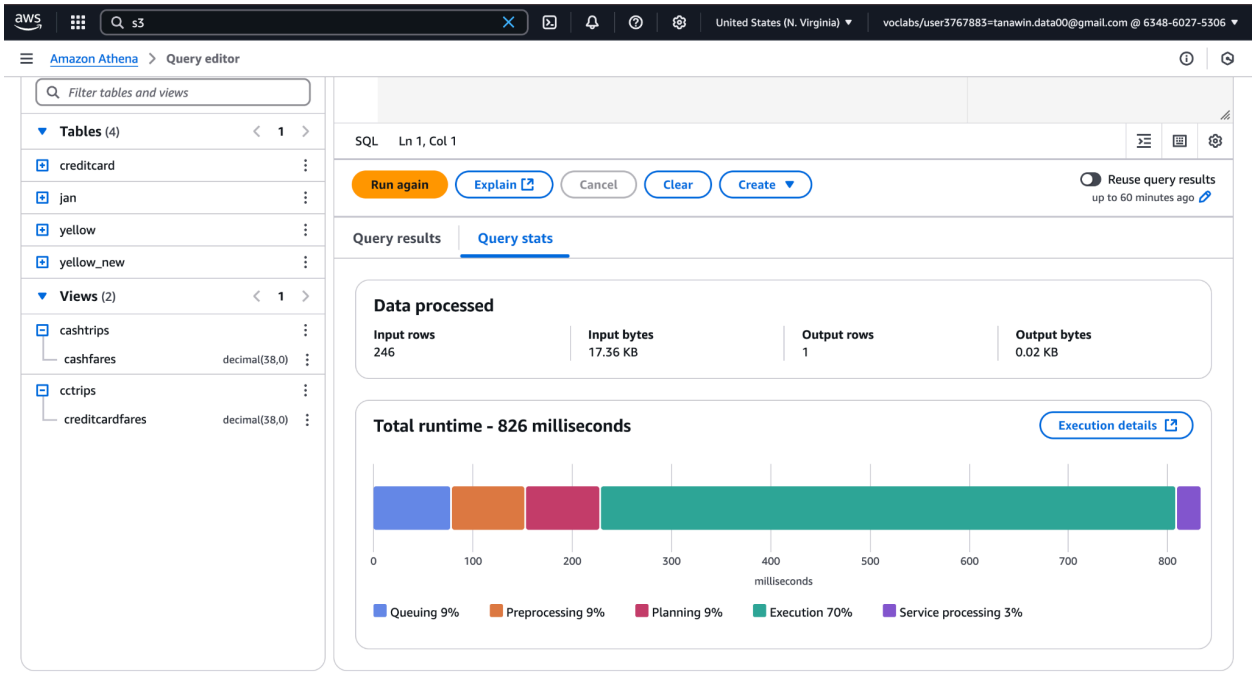
This figure shows results of performance fetched from cctrips.



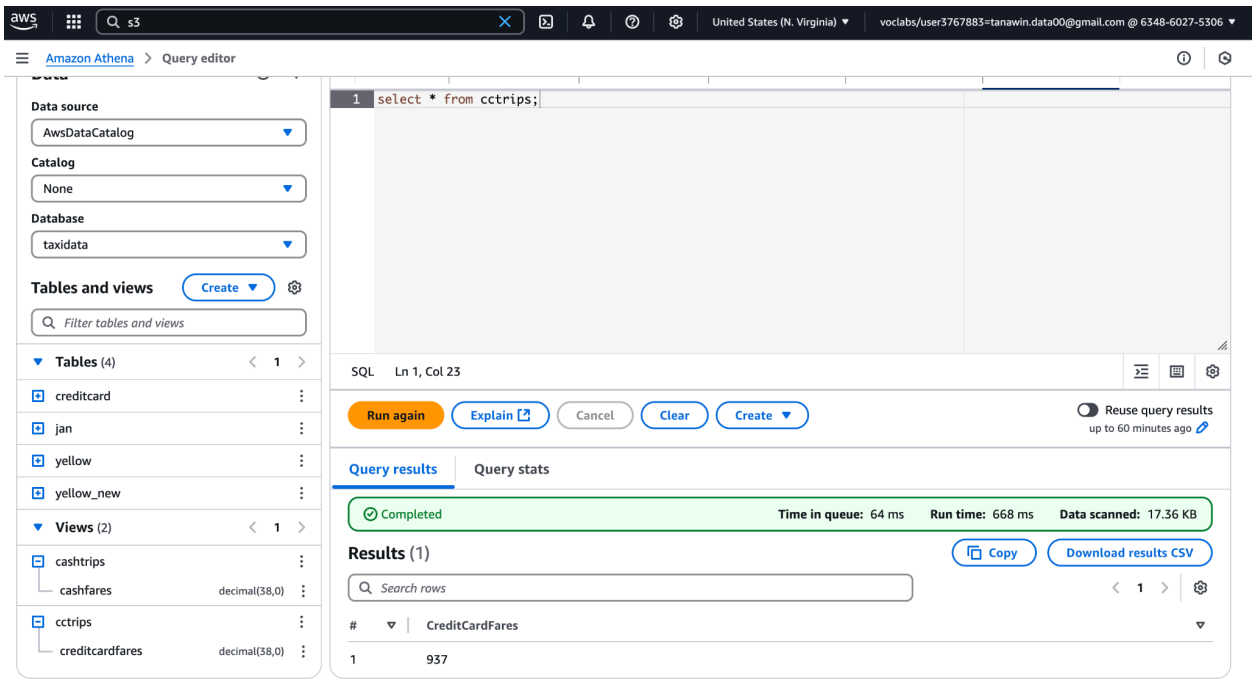
This figure shows results fetched from cashtrips.



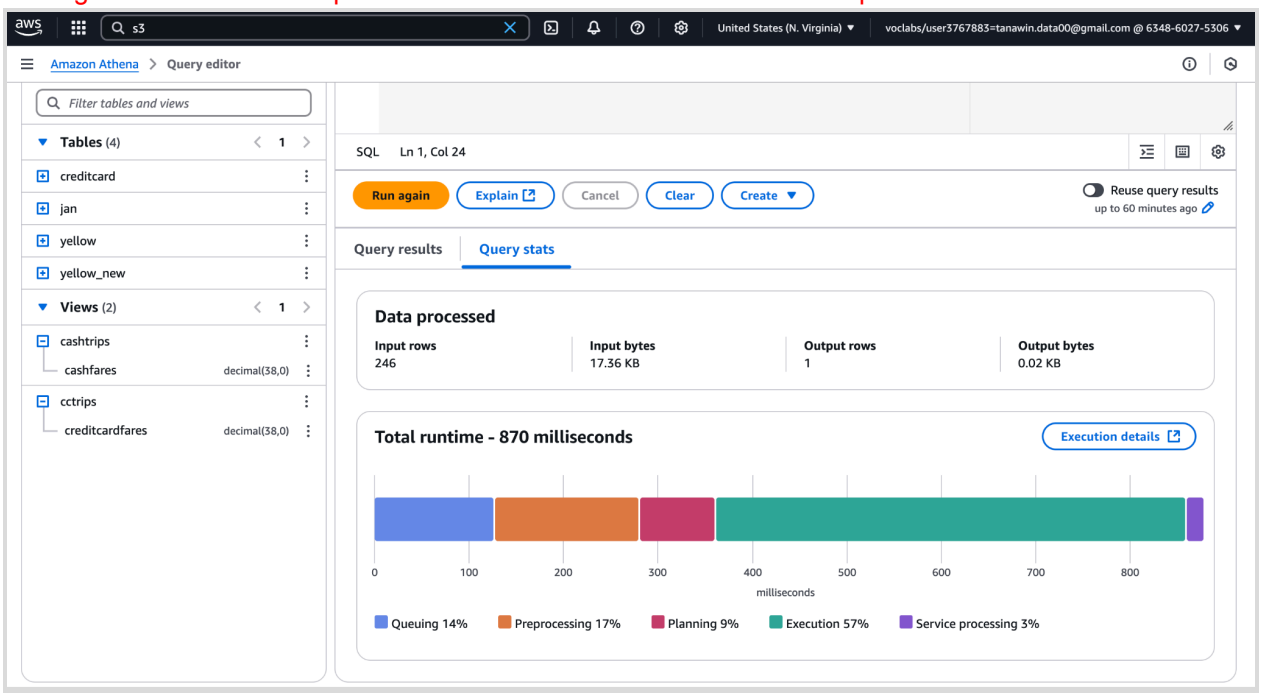
This figure shows results of performance fetched from cashtrips.



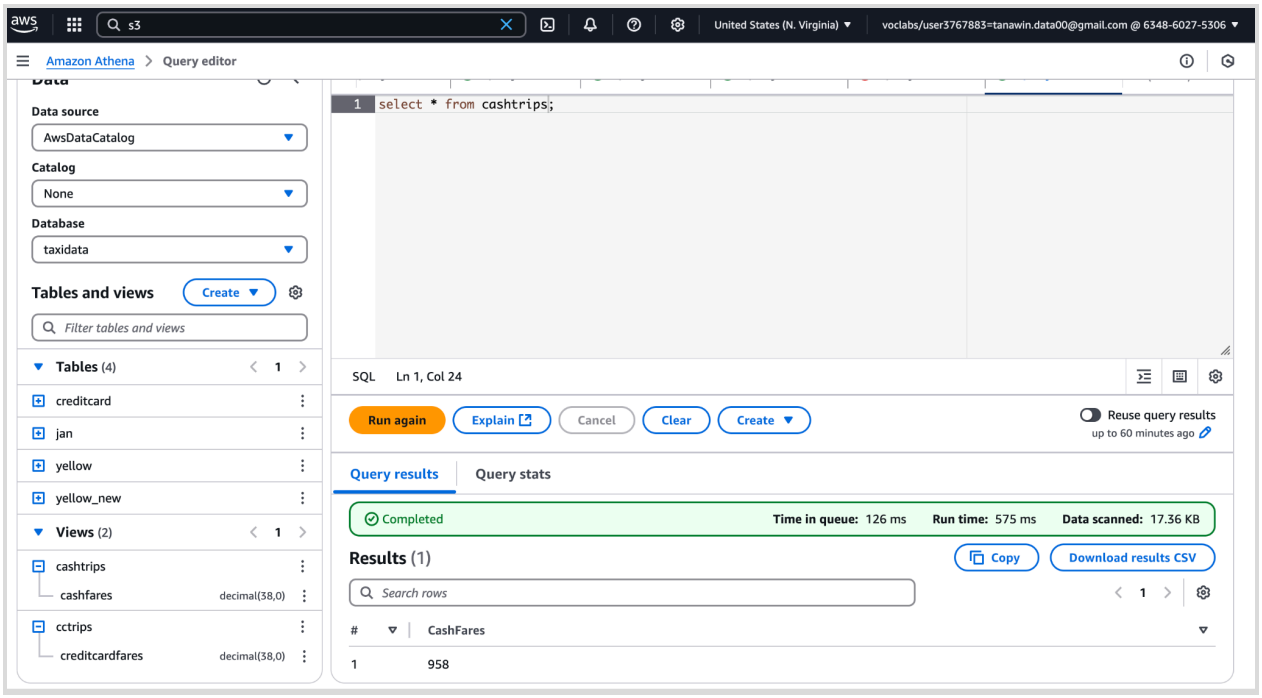
This figure shows results fetched from “select * from cctrips”.



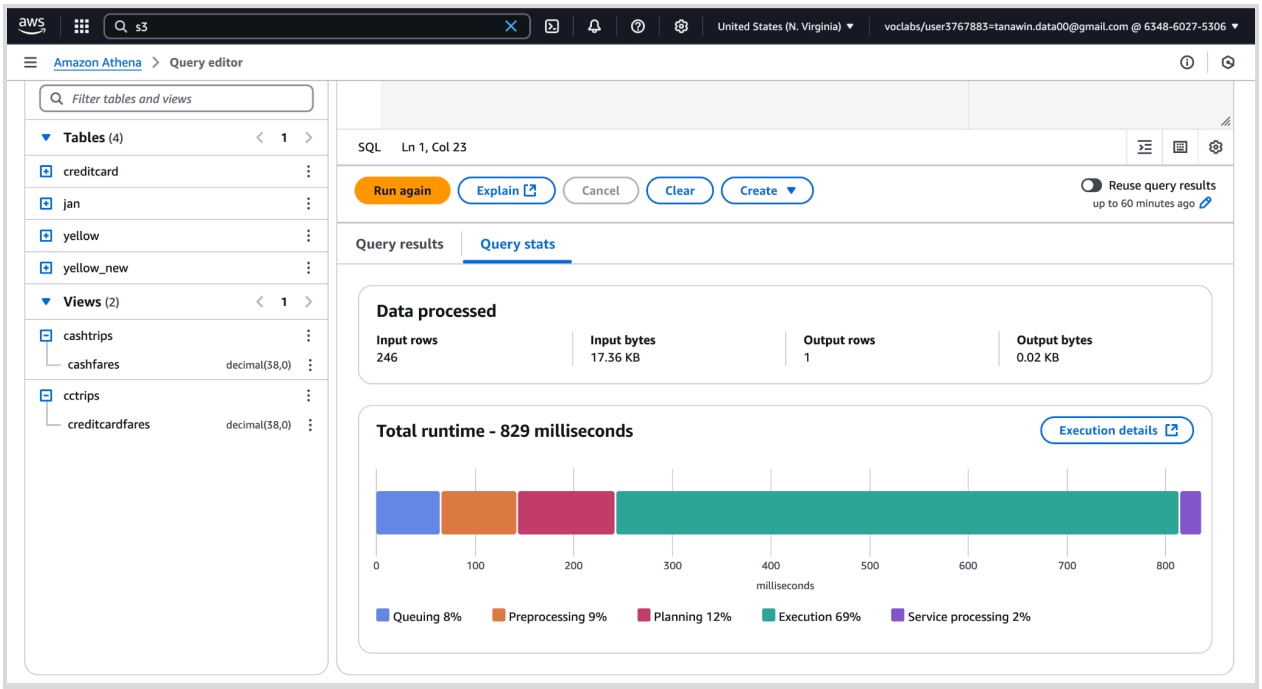
This figure shows results of performance fetched from “select * from cctrips”.



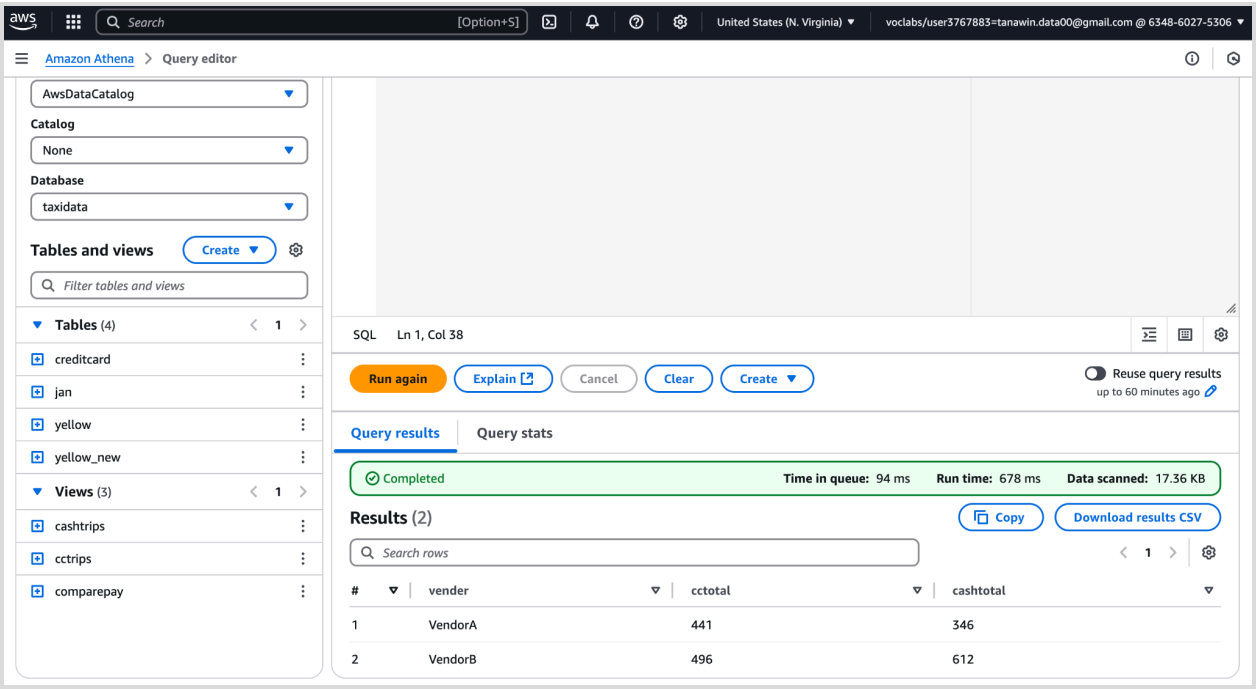
This figure shows results fetched from “select * from cashtrips”.



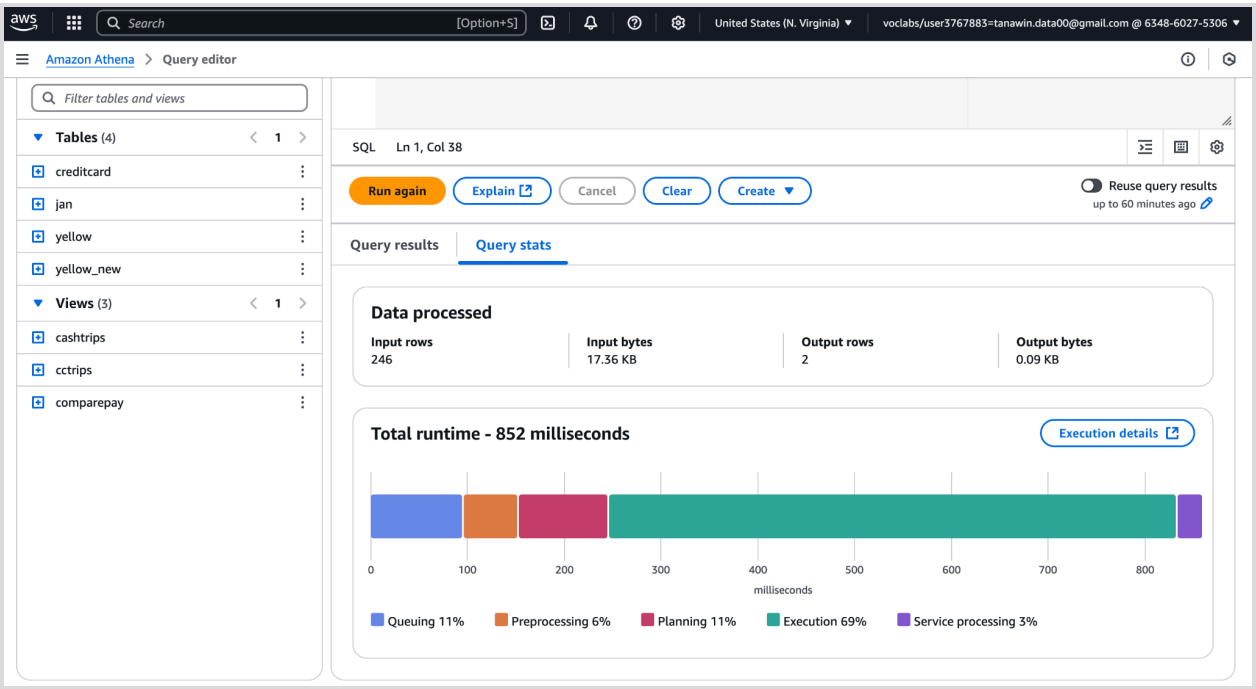
This figure shows results of performance fetched from “select * from cashtrips”.



This figure shows result from 19



This figure shows results from 19 query performance.



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.

AWS CloudFormation is a service that helps you model and set up your AWS resources so that you can spend less time managing those resources and more time focusing on your applications that run in AWS. You create a template that describes all the AWS resources that you want (like Amazon EC2 instances or Amazon RDS DB instances), and CloudFormation takes care of provisioning and configuring those resources for you. You don't need to individually create and configure AWS resources and figure out what's dependent on what; CloudFormation handles that. The following scenarios demonstrate how CloudFormation can help.

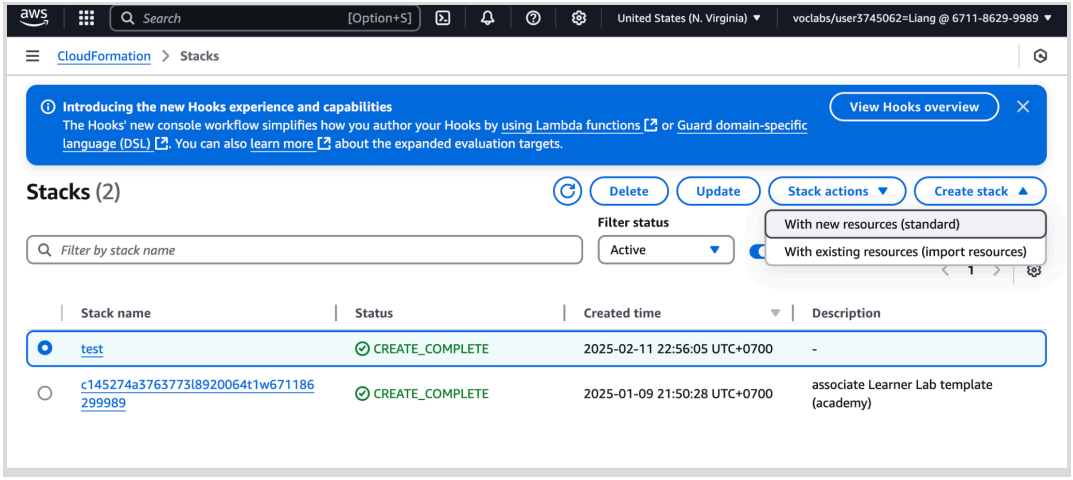
21. Review and run the example query.
Mary provides you with the following example query:

```
SELECT distance, paytype, fare, tip, tolls, surcharge, total FROM yellow
WHERE total >= 10.0 ORDER BY total DESC
```

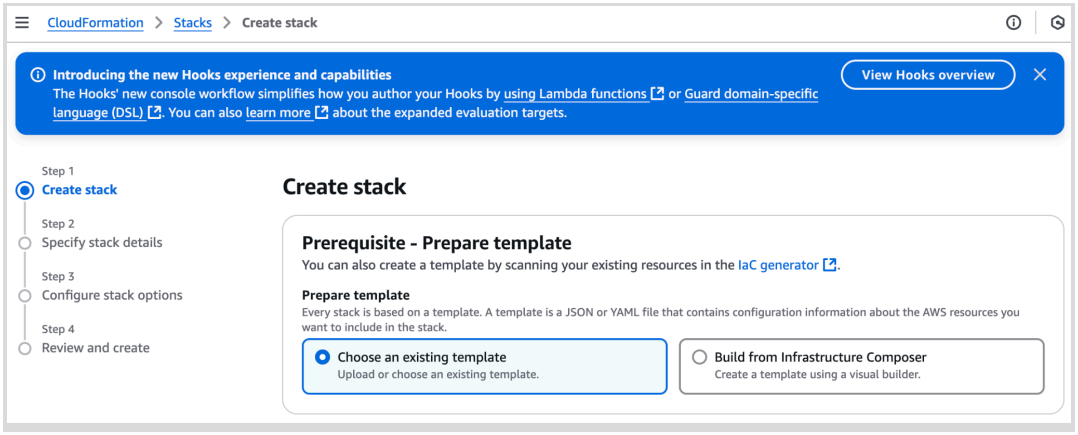
The template for using CloudFormation to connect Athena services as below:

```
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: "FaresOver10DollarsUS"
     QueryString: >
                  SELECT distance, paytype, fare, tip, tolls, surcharge,
total
                  FROM yellow
                  WHERE total >= 10.0
                  ORDER BY total DESC
Outputs:
 AthenaNamedQuery:
   Value: !Ref AthenaNamedQuery
```

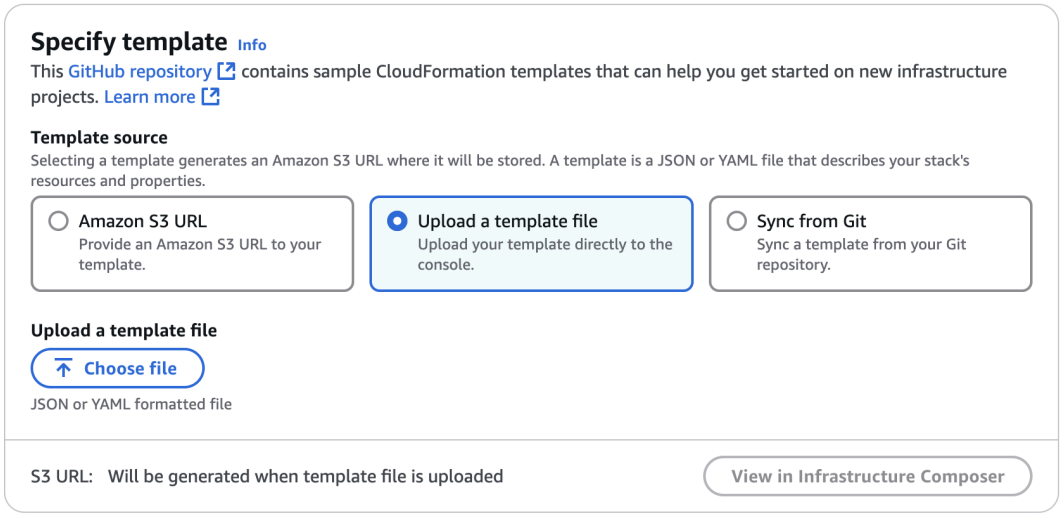
22. Select CloudFormation and then click Create stack -> With new resource (standard)



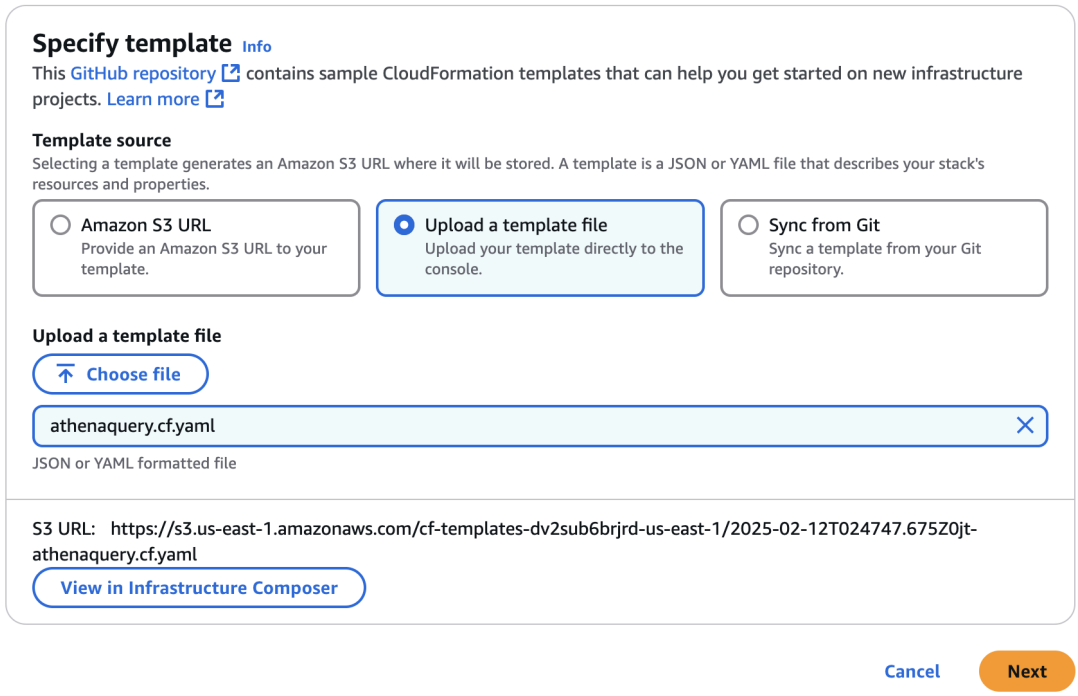
23. Step 1: Prerequisite - Prepare template
Select “Choose an existing template”



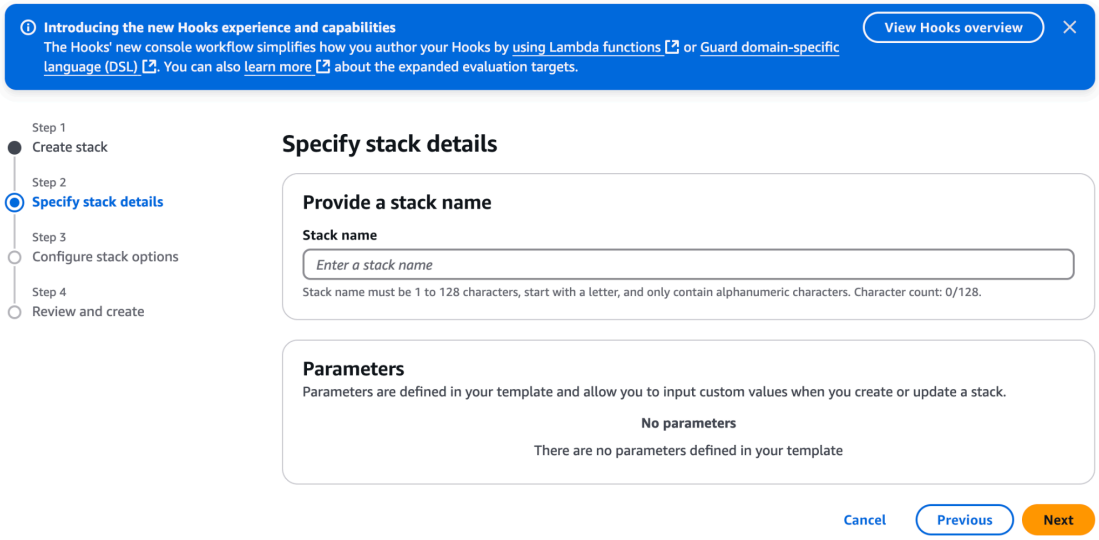
And select “Specify template” after that upload your template (YAML file)



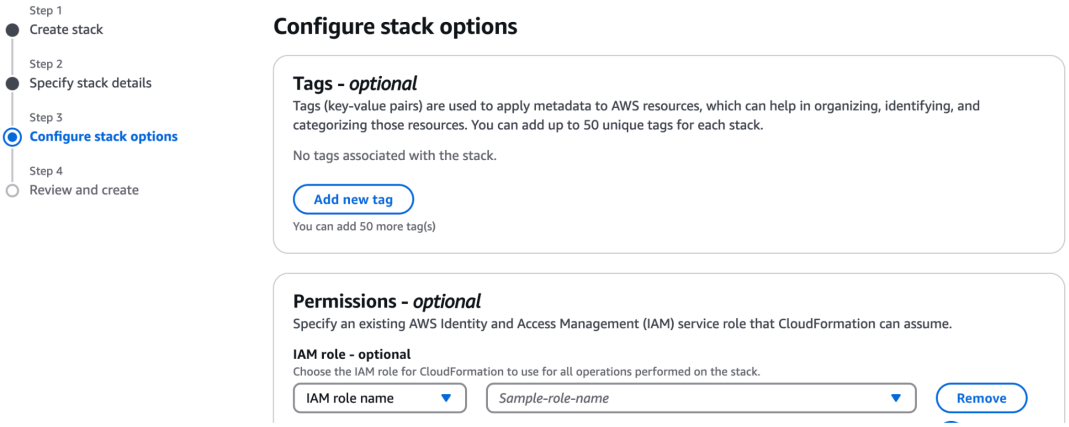
Then, press “Next”



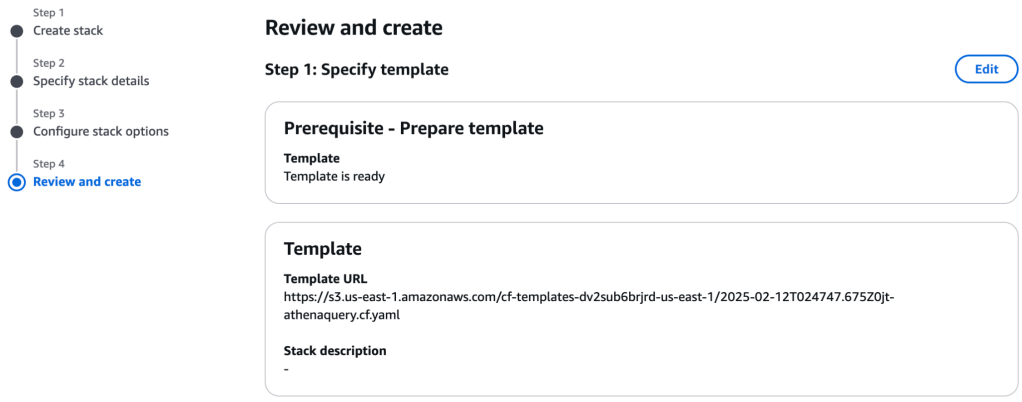
Step 2: Put your a stack name “queryforathena”



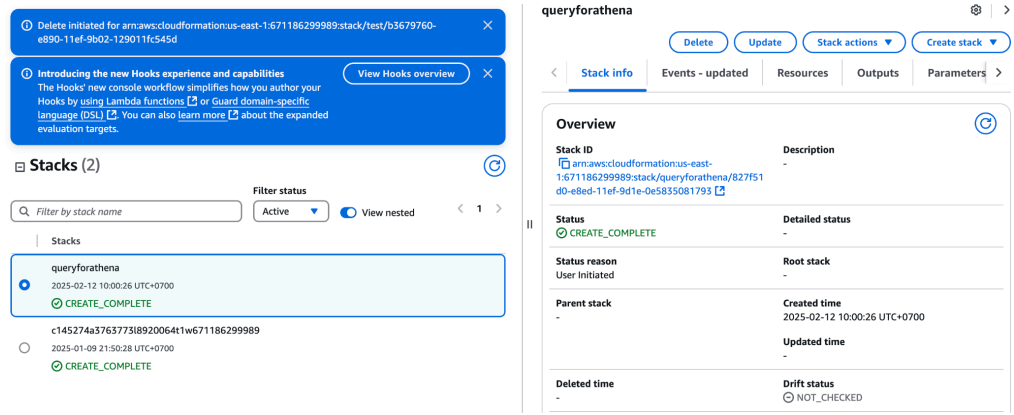
Step 3: Configure stack options - using default values.



Step 4: Review and create - using default values.



Finish



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

```
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:

  ```
  ~ $ aws athena get-named-query --named-query-id d50815cd-10fb-4b5a-b110-d9d8a3d8652a
  {
      "NamedQuery": {
          "Name": "FaresOver10DollarsUS",
          "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 >= 10.0 ORDER BY total DESC\n",
          "NamedQueryId": "d50815cd-10fb-4b5a-b110-d9d8a3d8652a",
          "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:

  ```
  ~ $ NQ=d50815cd-10fb-4b5a-b110-d9d8a3d8652a
  ~ $ echo $NQ
  d50815cd-10fb-4b5a-b110-d9d8a3d8652a
  ~ $
  ```

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.

Capture your result

This figure shows the query id from cloudformation.

This figure shows SQL query script from yaml config.

This figure shows the query id from NQ that was stored.