Optimizing data lakehouses with Starburst

Lab Guide (v3.1.1)

Table of Contents

Student setup 1
- Lab 1: Create student account 1

Starburst features 3
- Lab 1: Execute queries in Starburst Galaxy 3
- Lab 2: Exploring federated queries 21

Data lake tables 27
- Lab 1: Create a schema and tables 27
- Lab 2: Investigate Hive's special columns 41

Data lake performance 49
- Lab 1: Create tables with multiple file formats 49
- Lab 2: Using columnar file formats and eliminating small files 57
- Lab 3: Exploring table partitioning and bucketing 75

Table formats 88
- Lab 1: Explore the Delta Lake table format 88

Apache Iceberg 98
- Lab 1: Create and populate Iceberg tables 98
- Lab 2: Explore partitions with Iceberg 104
- Lab 3: Data modifications and snapshots with Iceberg 116

Advanced Iceberg 126
- Lab 1: Utilize Iceberg's MERGE statement 126
- Lab 2: Exercise advanced features of Iceberg 137

Cost-based optimizer 145
- Lab 1: The EXPLAIN command 145
- Lab 2: The EXPLAIN ANALYZE command 151
- Lab 3: Explore the impact of statistics on query plans 155

Access control 164
- Lab 1: Creating and validating RBAC policies with Starburst Galaxy 164
- Lab 2: Creating and validating ABAC policies with Starburst Galaxy 172

Data pipelines 187
- Lab 1: Construct a pipeline with insert-only transactions 187
- Lab 2: Construct a pipeline with the MERGE statement 207

Student setup

Lab 1: Create student account

Estimated completion time

- 5 minutes

Learning objectives

- This lab will guide you through the process of setting up your Starburst Galaxy student account. This account will be needed for future labs.

Prerequisites

- Before completing this lab, your instructor will send you an invitation email. If you have not received an email, please contact your instructor.

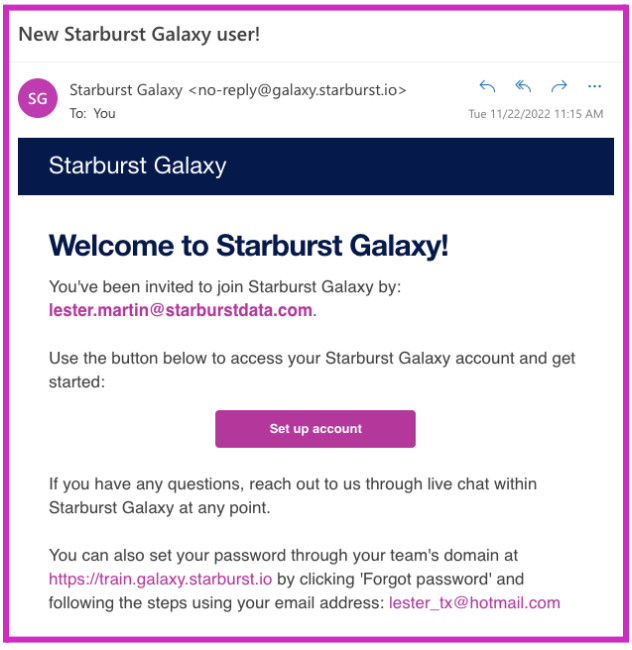
Activities

1. Set up account using invitation email
2. Create password
3. Confirm correct role

Step 1 - Set up account using invitation email

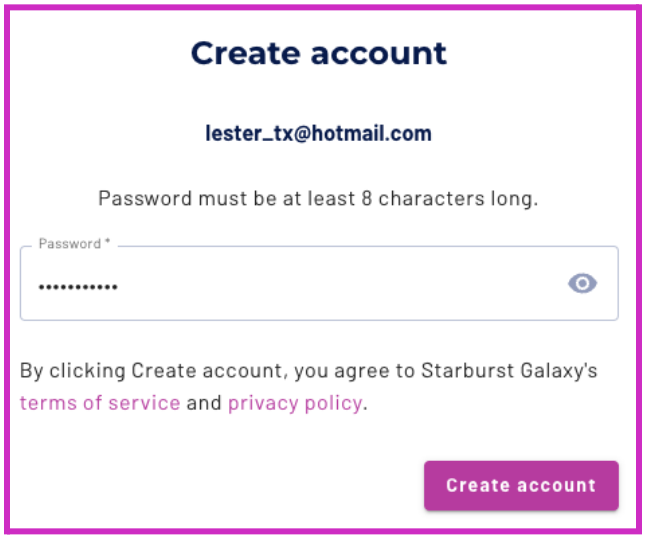
Your instructor has sent you an invitation to create a student account. The system will send an email similar to the one pictured below.

To get started, click the **Set up account** button.



Step 2 - Create password

Next, you must create an account password. Enter a password that meets the minimum requirements, then click the **Create account** button.



Step 3 - Confirm role

Starburst Galaxy users have an assigned role. For this module, your role should be set as `students`.

In the upper right corner of the screen, confirm that your role is set as `students`, as pictured below.

For all future labs, please ensure that your role is correct before proceeding. If your role is incorrect, please contact your instructor.

END OF LAB EXERCISE

Starburst features

Lab 1: Execute queries in Starburst Galaxy

Estimated completion time

- 40 minutes

Learning objectives

- This lab is designed to outline the basic query operations used in Starburst Galaxy. By the end of it, you will be able to sign in, navigate the UI, connect to data sources, and perform basic query operations like selecting, filtering, ordering, and grouping. Additional UI features are also introduced.

Prerequisites

- Student setup - Lab 1: Create student account

Activities

1. Sign in and verify role
2. Confirm clusters running
3. Navigate the cluster list
4. Select `sf1` schema from `tpch table`
5. Review Entity Relationship Diagram (ERD)
6. Access shared tables and execute `SELECT` query
7. Execute `DESCRIBE` query
8. Run `SELECT COUNT (*)` and `SELECT` * queries
9. Downloading results
10. Selecting specific columns
11. Filtering results by country or region
12. Ordering your query
13. Using the `AS` keyword
14. Grouping
15. Additional UI features

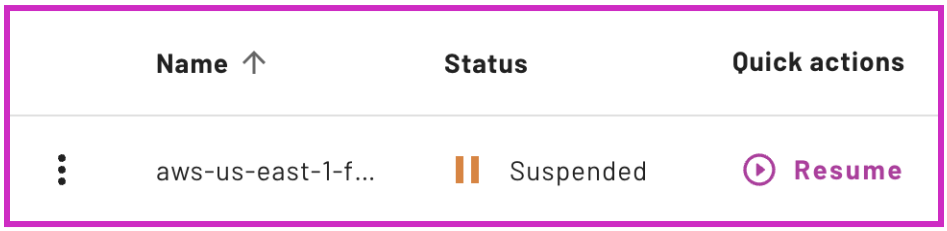
Step 1 - Sign in and verify role

Sign in and verify that the `students` role is selected in the upper-right corner of the screen.

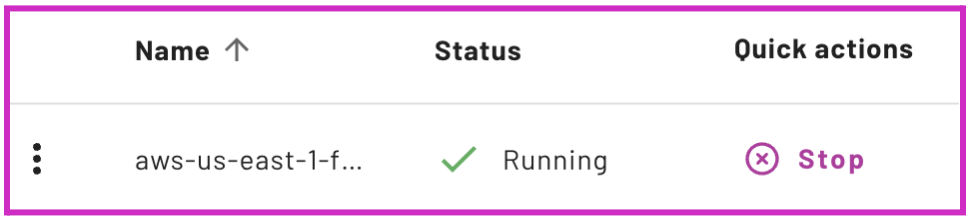
Step 2 - Confirm clusters running

Select **Clusters** in the navigation bar on the left.

Confirm that `aws-us-east-1-free` cluster instance is running. If it is not running, its **Status** will be listed as 'Suspended', as pictured below.

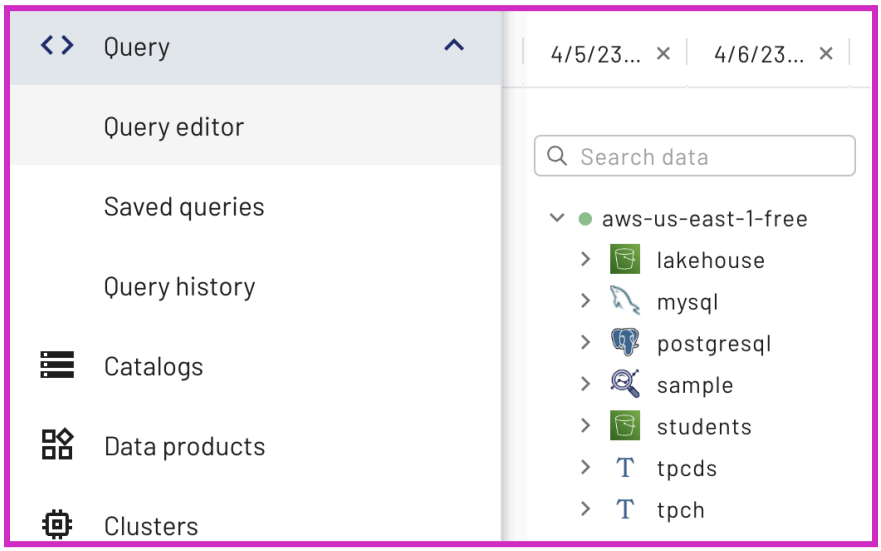


To restart the cluster, select **Resume** (or **Start**) from the **Quick actions** column. Confirm that the cluster is reporting a **Status** of `Running` before continuing, similar to the image below.



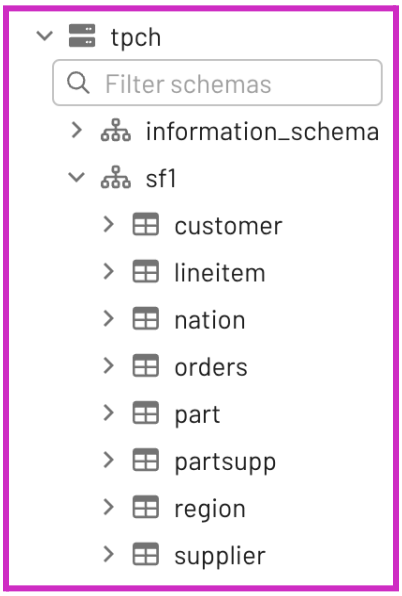
Step 3 - Navigate the cluster list

Navigate to **Query** > **Query editor** and then expand `aws-us-east-1-free` in the list of clusters.



Step 4 - Select `sf1` schema from `tpch` catalog

From the list of catalogs presented, expand `tpch` and then expand the `sf1` schema to see a list of tables.

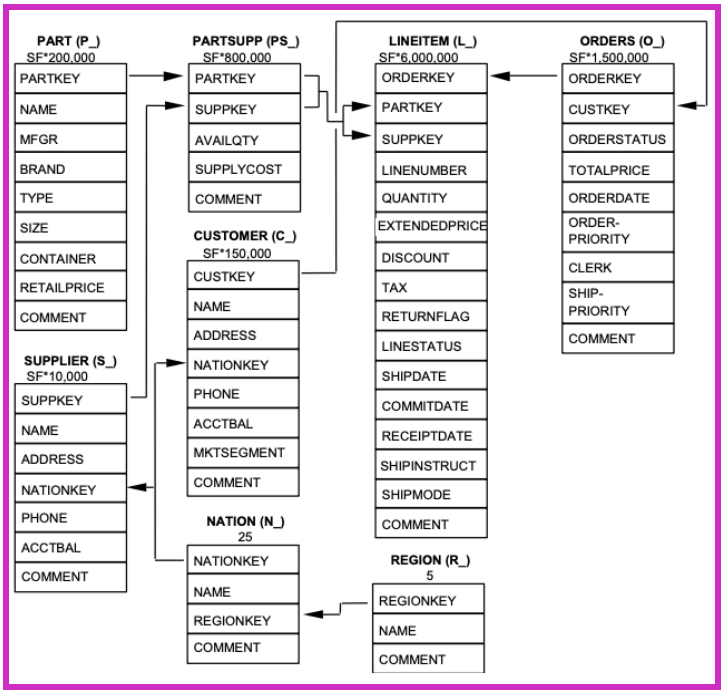


Step 5 - Review Entity Relationship Diagram (ERD)

The `tpch` catalog is an instance of the TCPH connector. As the documentation identifies, this is a deterministic data generator that has no actual backing datastore. The tables and their relationships are documented in the TPC-H Benchmark specification.

The Entity Relationship Diagram (ERD) presented below is from the benchmark specification. It describes the logical relationships between the tables. Review it and consider the relationships expressed.

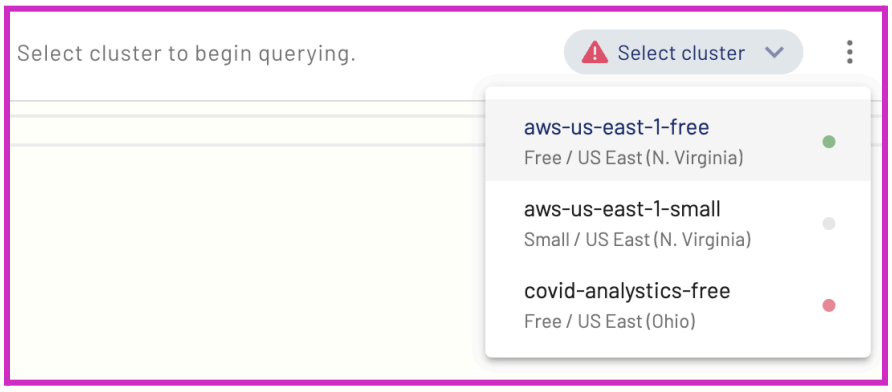
Note: Starburst web UI does NOT have a mechanism to generate an ERD like the following and it is only being presented to aid in understanding of this schema.



Step 6 - Access shared tables and execute `SELECT` query

The tables needed to complete this lab have been shared with you. To access them, click **Query**, then **Query editor**, and open a new tab in the editor by clicking on the **+** to the right of the last tab, similar to the image below.

Next, you need to ensure that the correct cluster is selected before you can run it. In the top right-hand corner of the screen, select the `aws-us-east-1-free (N. Virginia)` from the drop-down menu.



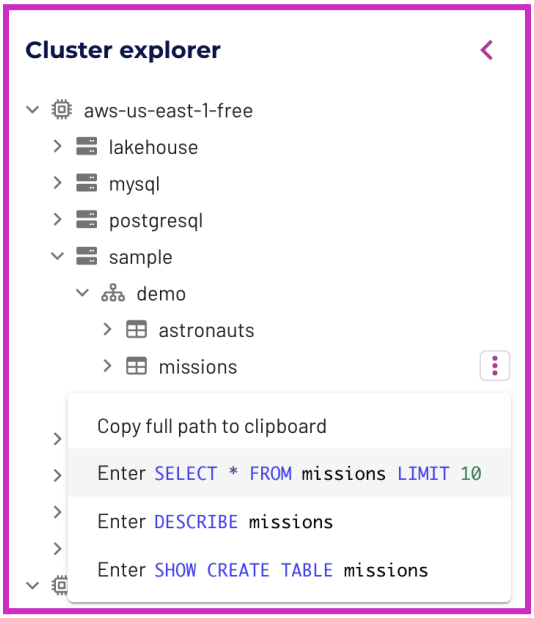
You now need to select the correct catalog, schema, and table before running the query.

Use the cluster list at the left of the editor pane and perform the following actions:

- Expand `aws-us-east-1-free` cluster.

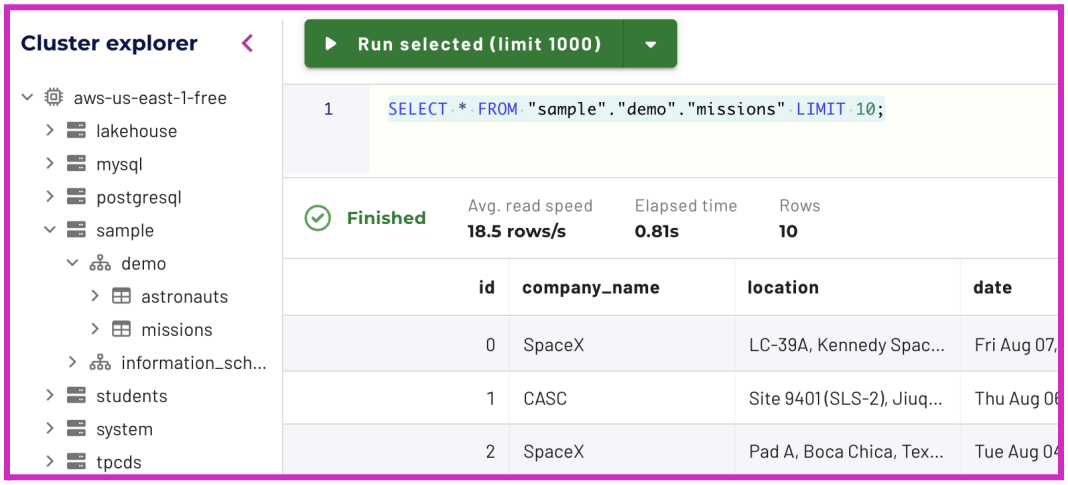
> **Note**: *The cluster should already be running. If it is not, it will pause for a few seconds before it can be used. This behavior is entirely normal.*

- Expand the `sample` catalog.
- Expand the `demo` schema.
- Hover over the `missions` table.
- Click the [pop-up vertical ellipsis menu] when it appears.
- Click the option showing the `SELECT` statement pictured below.



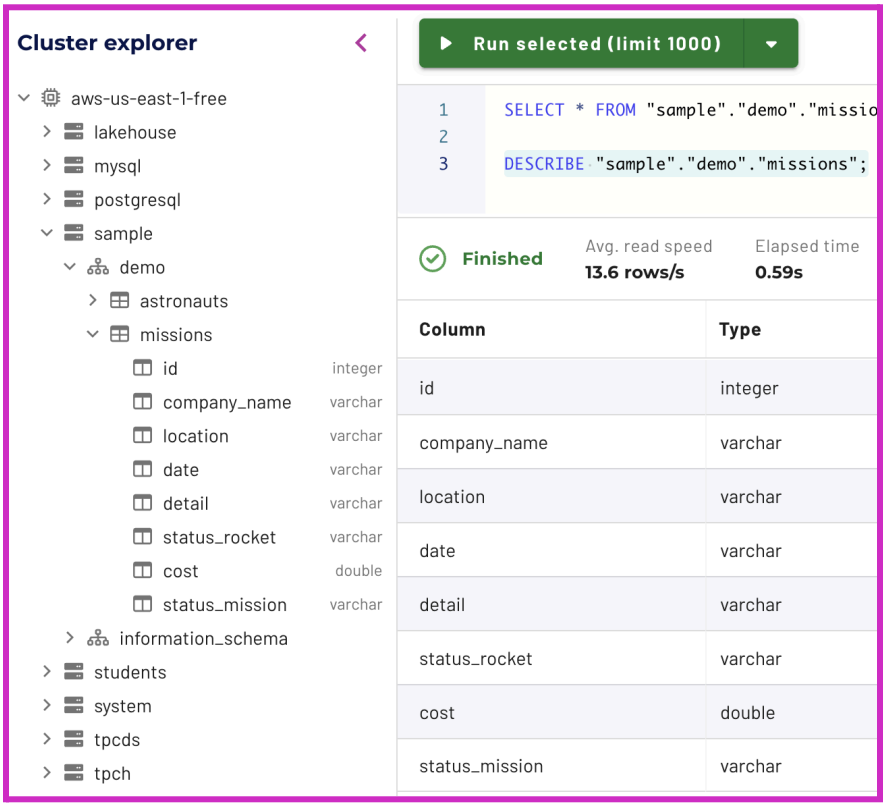
The query should now be copied into the query editor.

You are now ready to execute the query by selecting the **Run selected** button. The results will be displayed below the editor as pictured below.



Step 7 - Execute `DESCRIBE` query

Next, execute the `DESCRIBE` query using the same pop-up vertical ellipsis menu used in the last query. Compare the results with what you see when expanding the `missions` table in the cluster/catalog/schema list.



Step 8 - Run `SELECT COUNT (*)` and `SELECT *` queries

In the query editor, run a query on the `astronauts` table to verify there are 1279 rows present.

```
SELECT COUNT(*)
  FROM sample.demo.astronauts;
```

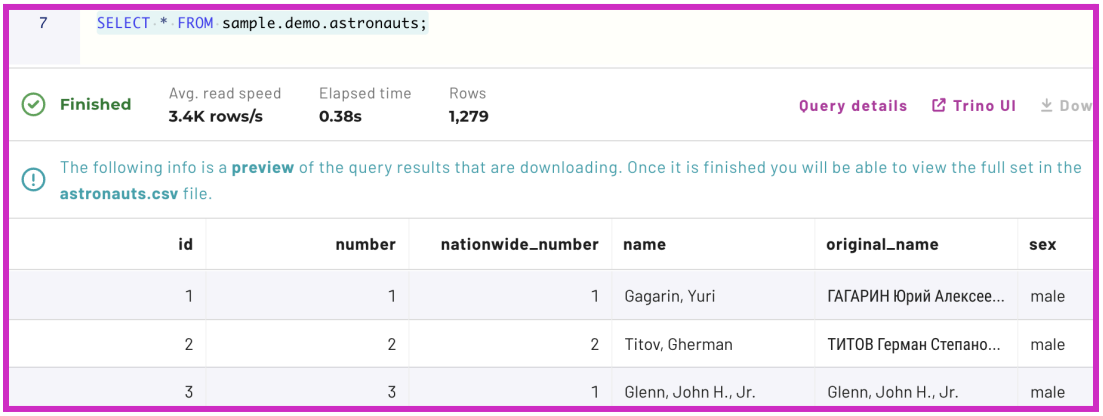
Then run a query to see the results.

```
SELECT *
  FROM sample.demo.astronauts;
```

Did you get all the rows?

Step 9 - Downloading results

The web UI limited the results to the first 1000 rows. If you want to display more rows, it's best to download the results in a file. Review the available Run Options and then download a file with all the rows. Were you able to get all of the rows?



Did you have an extra row? Why do you think that might be there?

The extra row is due to the downloaded file containing a header row.

Note: Generally speaking, the **Query editor** is designed for query results that can be limited. The **Run and download** option is convenient, but it only supports CSV files. If you want to save in another format and/or you are creating a new dataset for further processing, there are better options. Inserting the results into a new or existing table will give you lots of flexibility in this situation. This is discussed more in the Data Pipeline module.

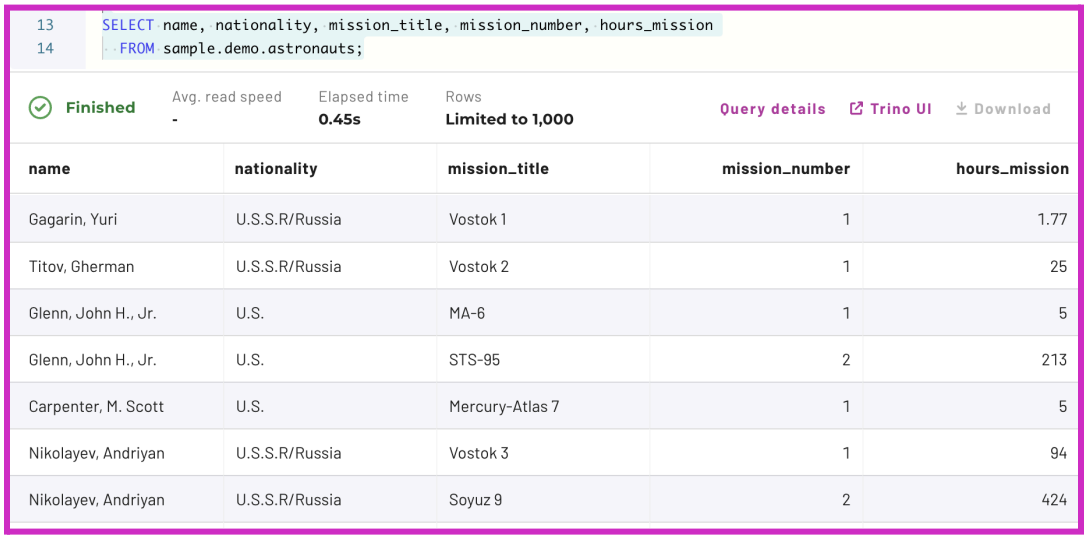
Back in the **Query editor**, explore the columns of `sample.demo.astronauts` along with some of the data in the results pane.

```
DESCRIBE sample.demo.astronauts;
```

Step 10 - Selecting specific columns

Select just a few specific columns to help review the data better.

```
SELECT name, nationality, mission_title, mission_number,
hours_mission
  FROM sample.demo.astronauts;
```



name	nationality	mission_title	mission_number	hours_mission
Gagarin, Yuri	U.S.S.R/Russia	Vostok 1	1	1.77
Titov, Gherman	U.S.S.R/Russia	Vostok 2	1	25
Glenn, John H., Jr.	U.S.	MA-6	1	5
Glenn, John H., Jr.	U.S.	STS-95	2	213
Carpenter, M. Scott	U.S.	Mercury-Atlas 7	1	5
Nikolayev, Andriyan	U.S.S.R/Russia	Vostok 3	1	94
Nikolayev, Andriyan	U.S.S.R/Russia	Soyuz 9	2	424

As you can see in the previous results, there are multiple records for the same name instead of a single row for each astronaut. For example, John Glenn is included in the third and fourth rows.

Each record represents details about unique trips to space by the astronauts (for example, the number of hours the astronaut was on a specific mission). A possible more descriptive table name could have been `astronaut_missions`.

Step 11 - Filtering results by country or region

As you scroll through your results, you will notice that astronauts come from many different countries. It would be good to see results listed by country of origin or even to exclude one country from a list. For instance, we might want to see all of the non-U.S. results outlined specifically.

To do this, apply the following filter to your query referencing the `nationality` column.

```
SELECT name, nationality, mission_title, mission_number,
hours_mission
  FROM sample.demo.astronauts
 WHERE nationality != 'U.S.';
```

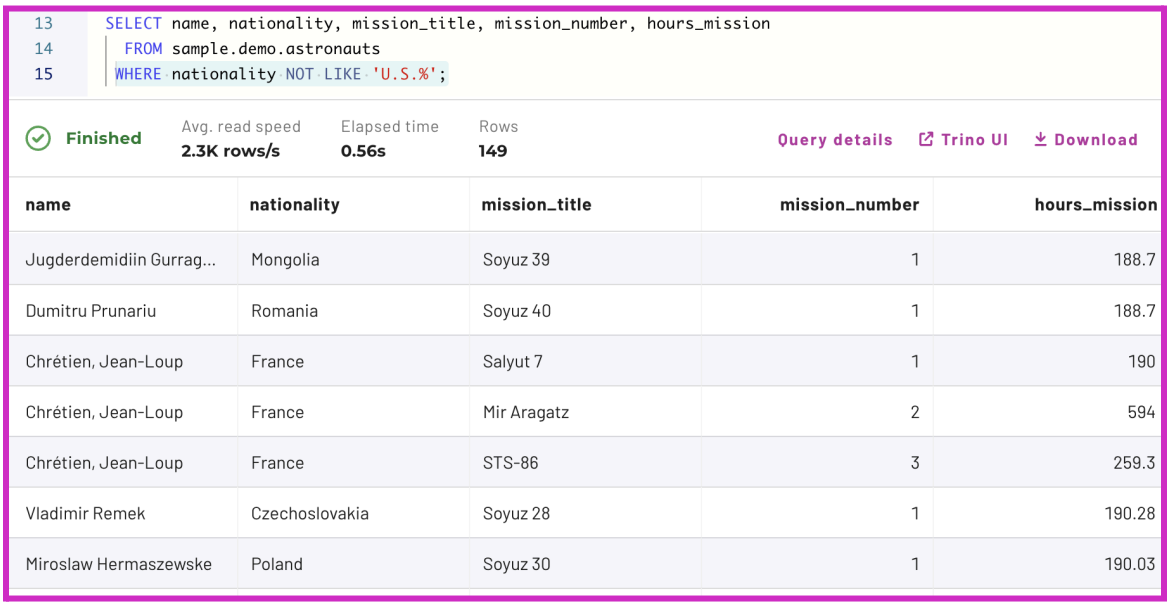
This query eliminates records with a `nationality` listed as “U.S.”. It will filter out only records from the United States.

Now imagine that you wanted to eliminate records that included any `nationality` that included the letters “U.S.” in the title. This would include the United States, but it would also include the former U.S.S.R. because its name also includes the letters “U.S.” in the title.

To do this, we can introduce the wildcard character “%,” which broadens the filter to include any results with the specified characters in the `nationality` field. Replace your previous code with the code below and check the results.

```
SELECT name, nationality, mission_title, mission_number,
hours_mission
  FROM sample.demo.astronauts
 WHERE nationality NOT LIKE 'U.S.%';
```


Records from the “U.S.” and “U.S.S.R.” have both been filtered out of the result.



```
13  SELECT name, nationality, mission_title, mission_number, hours_mission
14    FROM sample.demo.astronauts
15  WHERE nationality NOT LIKE 'U.S.%';
```

Finished | Avg. read speed 2.3K rows/s | Elapsed time 0.56s | Rows 149 | Query details | Trino UI | Download

name	nationality	mission_title	mission_number	hours_mission
Jugderdemidiin Gurrag...	Mongolia	Soyuz 39	1	188.7
Dumitru Prunariu	Romania	Soyuz 40	1	188.7
Chrétien, Jean-Loup	France	Salyut 7	1	190
Chrétien, Jean-Loup	France	Mir Aragatz	2	594
Chrétien, Jean-Loup	France	STS-86	3	259.3
Vladimir Remek	Czechoslovakia	Soyuz 28	1	190.28
Miroslaw Hermaszewske	Poland	Soyuz 30	1	190.03

Step 12 - Ordering your query

To aid in your review, you can add an `ORDER BY` clause to your SQL statement to order the remaining records by `nationality`. Further refinement of the ordering can be accomplished by including a second column name in your `ORDER BY` clause.

```
SELECT name, nationality, mission_title, mission_number,
hours_mission
  FROM sample.demo.astronauts
 WHERE nationality NOT LIKE 'U.S.%'
 ORDER BY nationality, name;
```

```
13  SELECT name, nationality, mission_title, mission_number, hours_mission
14    FROM sample.demo.astronauts
15   WHERE nationality NOT LIKE 'U.S.%'
16   ORDER BY nationality, name;
```

Finished | Avg. read speed 2.9K rows/s | Elapsed time 0.44s | Rows 149 | Query details | Trino UI | Download

name	nationality	mission_title	mission_number	hours_mission
Mohmand, Abdul Ahad	Afghanistan	3	1	212
Thomas, Andrew S. W.	Australia	STS-77	1	240
Thomas, Andrew S. W.	Australia	24	2	3384
Thomas, Andrew S. W.	Australia	STS-102	3	307
Thomas, Andrew S. W.	Australia	STS-114	4	333
Viehbock, Franz	Austria	Soyuz TM13/12	1	190.2
De Winne, Frank	Belgium	20	2	4508.6

Until this point, your queries have returned individual rows. You will now use an aggregate function to summarize your results into a single value. Review the results from your previous query. Run the following SQL statement to calculate the number of rows contained in the `astronauts` table and return the maximum and minimum values within the `hours_mission` column. ***The results should indicate there are 149 rows in this table.***

```
SELECT COUNT(), MAX(hours_mission), MIN(hours_mission)
  FROM sample.demo.astronauts
 WHERE nationality NOT LIKE 'U.S.%';
```

```
19  SELECT COUNT(), MAX(hours_mission), MIN(hours_mission)
20    FROM sample.demo.astronauts
21   WHERE nationality NOT LIKE 'U.S.%';
```

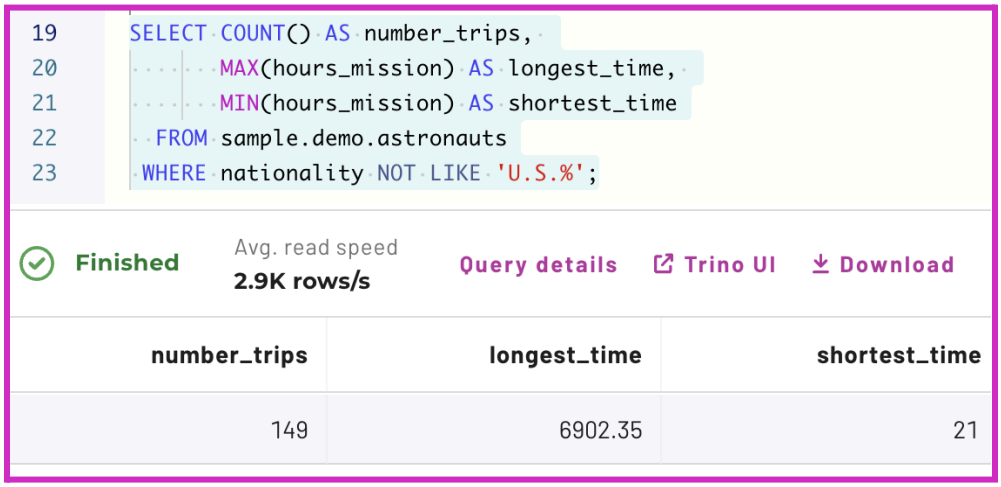
Finished | Avg. read speed 3K rows/s | Query details | Trino UI | Download

_col0	_col1	_col2
149	6902.35	21

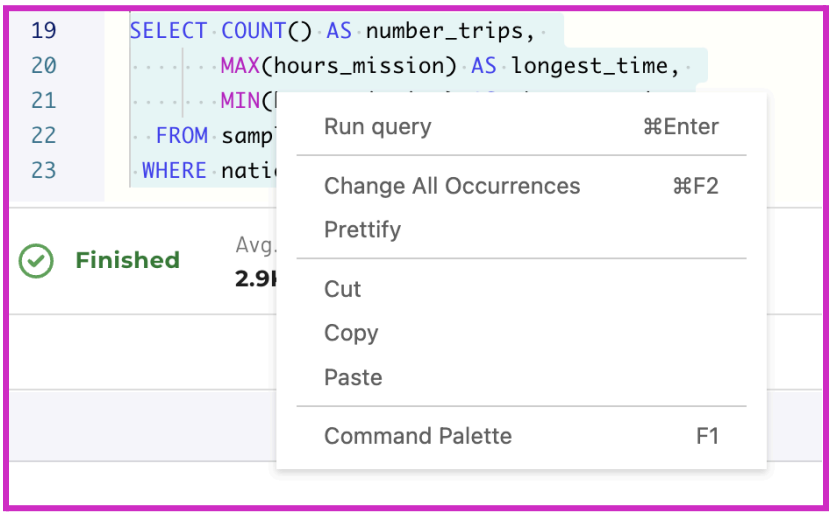
Step 13 - Using the AS keyword

You've aggregated your results, but the column names still do not provide enough information to understand what the values represent. You will improve the clarity of your query results by introducing column aliases using the AS keyword.

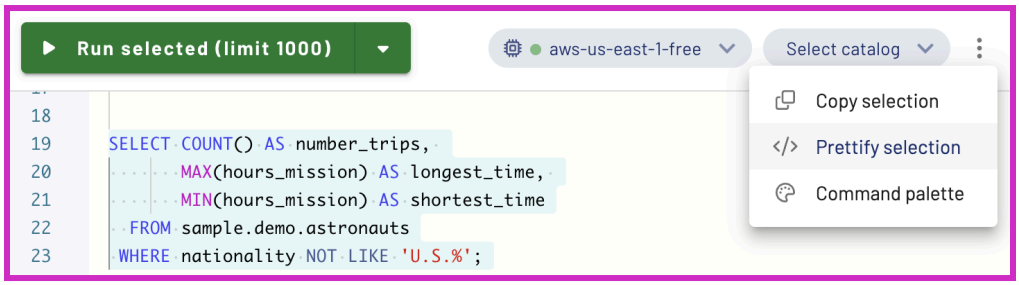
```
SELECT COUNT(*) AS number_trips,
       MAX(hours_mission) AS longest_time,
       MIN(hours_mission) AS shortest_time
  FROM sample.demo.astronauts
 WHERE nationality NOT LIKE 'U.S.%';
```



The Starburst web UI has a feature called "Prettify". You can access Prettify by highlighting your query and right-clicking. This will open a pop-up context menu.



You can also use Prettify by clicking on the editor pane vertical ellipsis menu near the upper-right corner of the UI.



Regardless of which way you exercise this feature, your query should now look similar to this.

```
SELECT
 COUNT() AS number_trips,
 MAX(hours_mission) AS longest_time,
 MIN(hours_mission) AS shortest_time
FROM
 sample.demo.astronauts
WHERE
 nationality NOT LIKE 'U.S.%';
```

Step 14 - Grouping

Aggregate functions are also useful in conjunction with a `GROUP BY` clause. Update your query to present the records by `nationality`. Once you've successfully returned the records organized by `nationality`, add sorting options to help you analyze the results. Your query may appear as follows. The changes from the prior presented query are highlighted below.

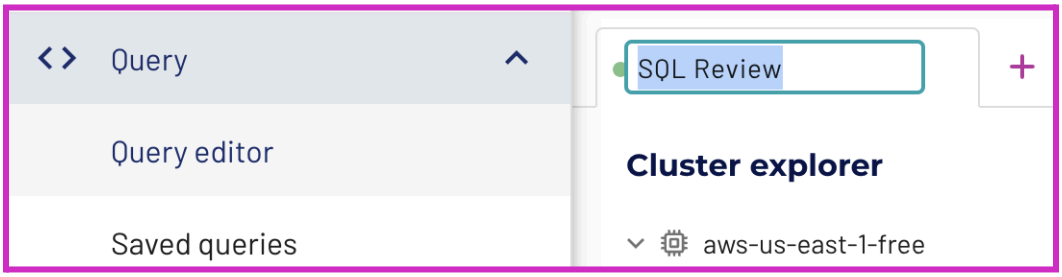
```
SELECT
 Nationality, COUNT() AS number_trips,
 MAX(hours_mission) AS longest_time,
 MIN(hours_mission) AS shortest_time
FROM
 sample.demo.astronauts
WHERE
 nationality NOT LIKE 'U.S.%'
GROUP BY
 nationality
ORDER BY
 number_trips DESC,
 longest_time DESC;
```

In the results you received, verify the following.

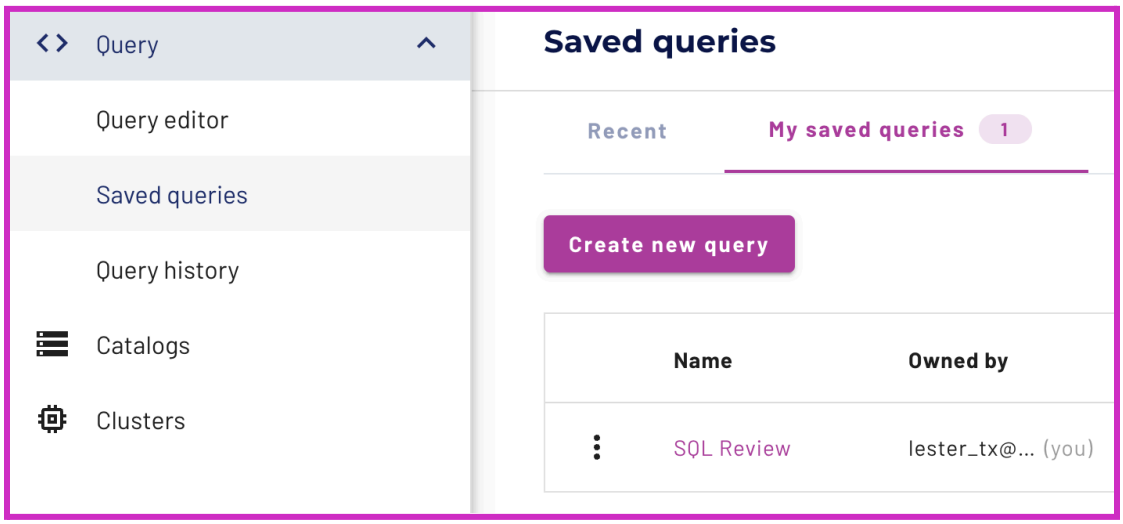
- Japan had the most number of trips with 20.
- There are ~ 20 nations that tied with a single trip each.

Step 15 - Additional UI features

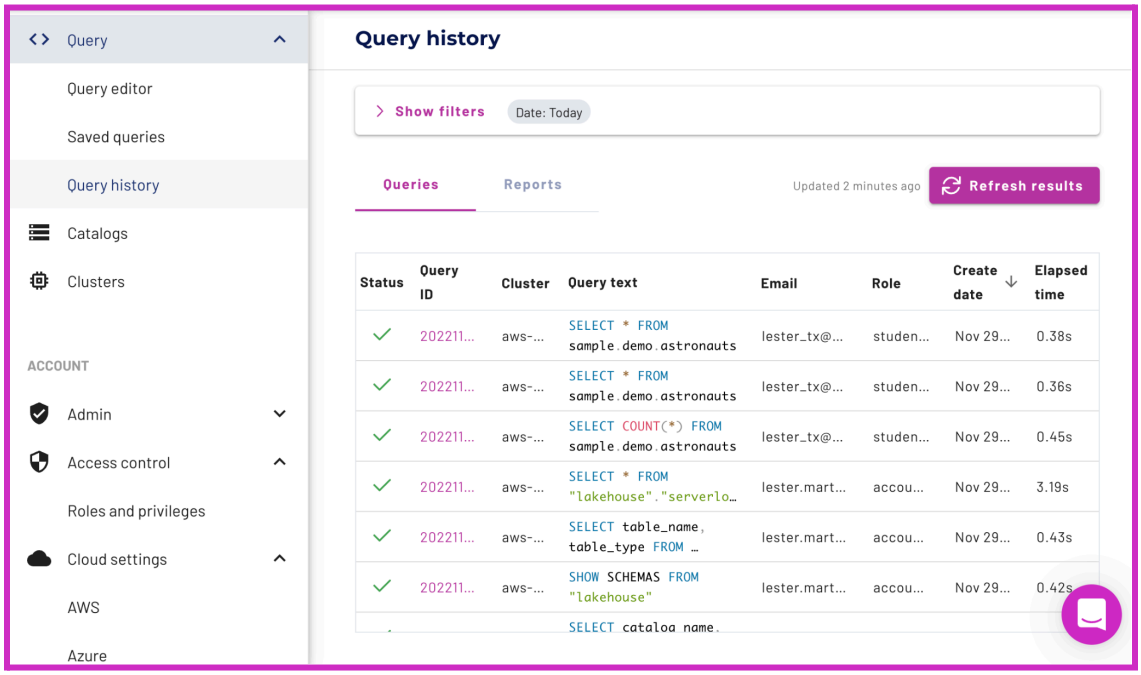
The web UI has additional features to help in creating, executing, and organizing SQL statements. Click the name of the active tab above (*it should be a timestamp unless previously edited*) and change it to `SQL Review`.



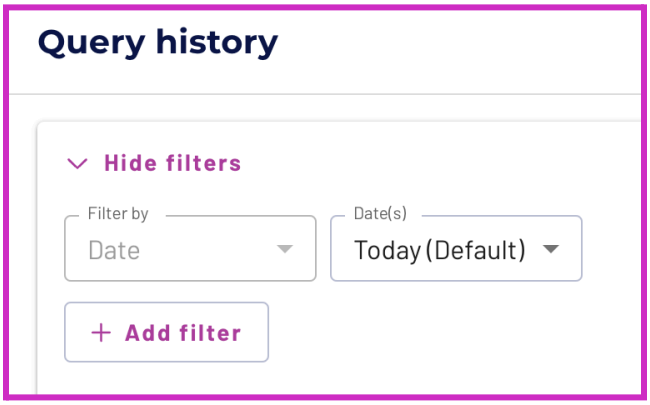
Navigate to **Query** > **Saved queries** to verify `SQL Review` is present under **My saved queries**.



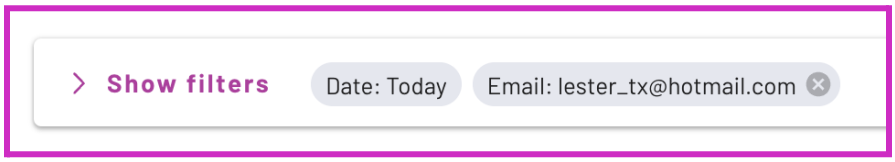
Navigate to **Query** > **Query history** to see the queries that have been recently executed.



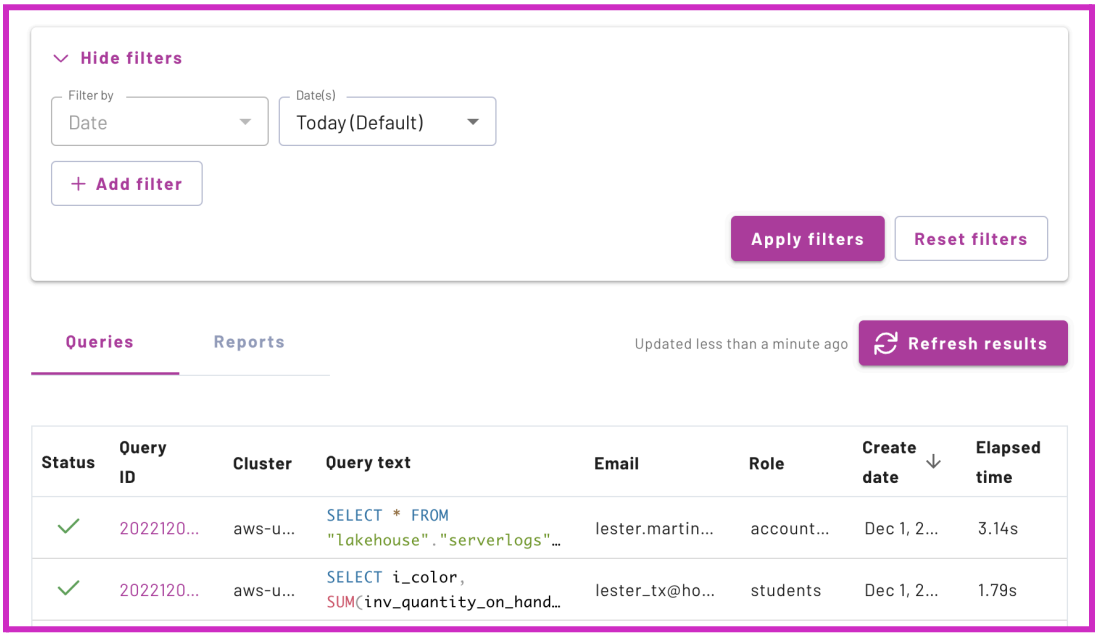
This shows queries across all users. click **Show filters** from above, then use the **Add filter** button (below) to limit the results to your queries.



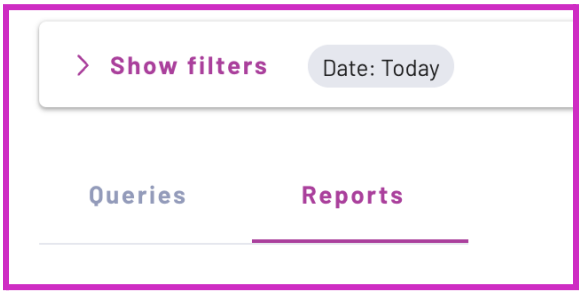
Your search bar should list all applied filters once you **Hide filters**.



Feel free to experiment with additional filters. Share anything you find interesting with the class or the instructor. When done, click **Reset filters** to see all **Queries**.

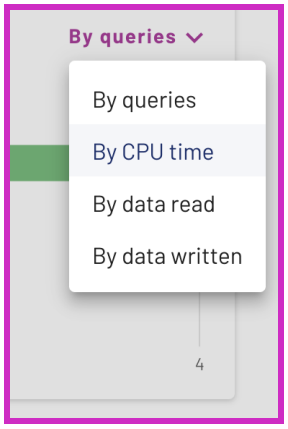


Move away from the **Queries** tab by clicking on **Reports**.



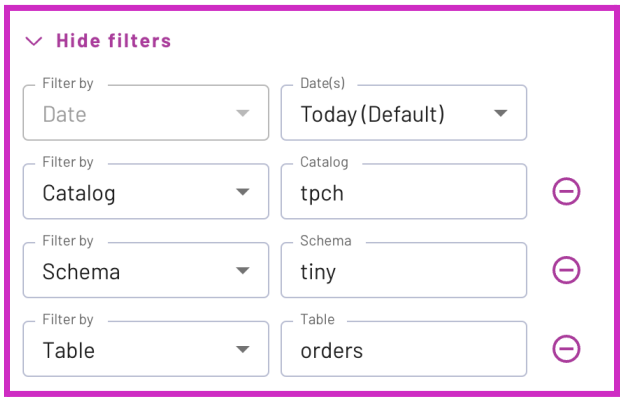
What is the highest number of **Queries over time**? What times were the peaks?

Review the **Top users** report to determine who has run the most queries. How many queries did this user run? If you were not the top user, how many queries did you run? Did the top user change when you toggled **By queries** to **By CPU time** on the right side of this report?

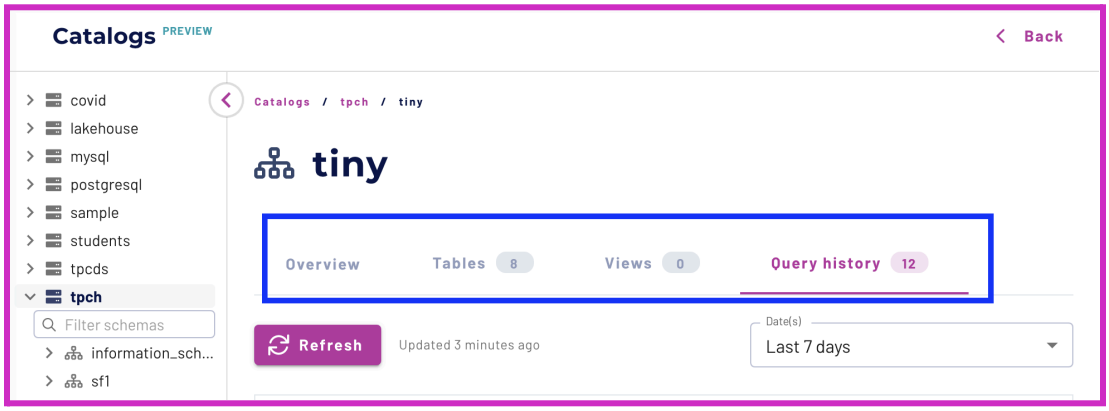


Does the **Top tables** report seem accurate based on the queries run in this lab? Re-add the filter to limit results to the queries you've run. Are the results still similar?

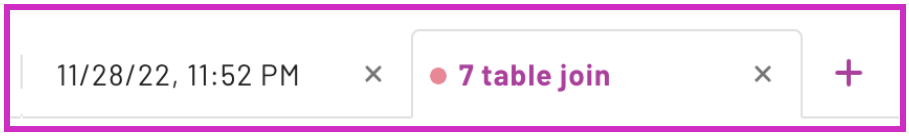
You may have noticed you could easily isolate historical results to specific catalogs, schemas, and tables from your earlier exploration of the **Query history** filters.



Navigate to **Catalogs** from the left navigation. Expand `tpch` to show its schemas. Explore the available features to find reports similar to those found in **Query History** for some of the schema(s) and table(s) used in this lab.



The instructor has shared a query with you previously. To access it, click **Query**, then **Saved queries**, and finally the **Shared with me** tab. Open the `7 table join` query in the list.



Confirm that both of the queries inside this shared artifact run successfully.

END OF LAB EXERCISE

Lab 2: Exploring federated queries

Estimated completion time

- 15 minutes

Learning objectives

- This lab will take the querying skills used in previous labs and apply them to federated queries. Emphasis will be on showcasing how federation works, what it allows you to do, and how to incorporate it into your workflows.

Prerequisites

- Student setup - Lab 1: Create student account

Activities

1. Sign in and verify cluster
2. Query `postgresql` catalog and `tcph_sf1` schema
3. View query details
4. Query orders table using projection
5. Query customer table
6. Execute federated query joining two tables
7. Federate across three tables

Step 1 - Sign in and verify cluster

Sign in and verify that the `students` role is selected in the upper-right corner of the screen.

Navigate to the **Clusters**. Ensure `aws-us-east-1-free` is reporting a **Status** of Running before continuing.

Step 2 - Query `postgresql` catalog and `tcph_sf1` schema

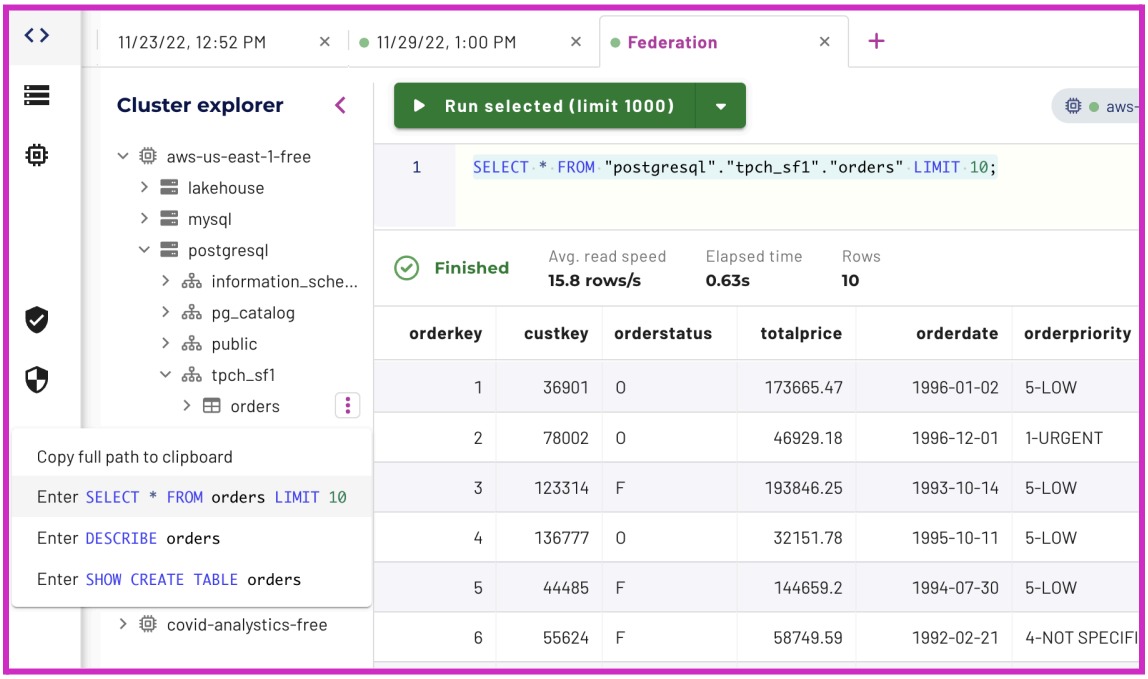
Navigate to **Query** > **Query editor**. Expand `aws-us-east-1-free` in the cluster list.

Open a new tab and rename it as `Federation`.

Now, expand the `postgresql` catalog. Expand the `tpch_sf1` schema. Locate the `orders` table. Run a query to see ten rows using the `SELECT` command.

```
SELECT * FROM postgresql.tpch_sf1.orders LIMIT 10;
```

The results will look similar to the image below.

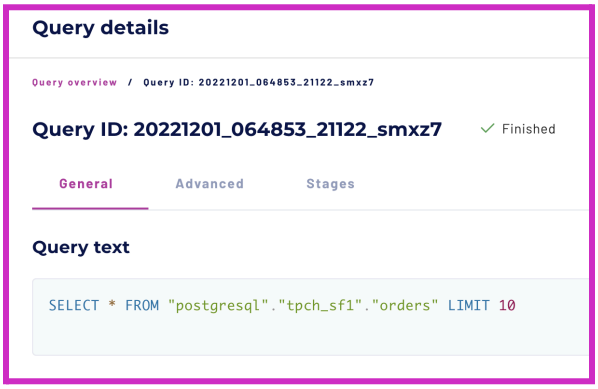


Sometimes the origin of data is unclear. Where do you think this data came from? The `tpch_sf1` schema name suggests that this is a copy of the `tpch.sf1.orders` table.

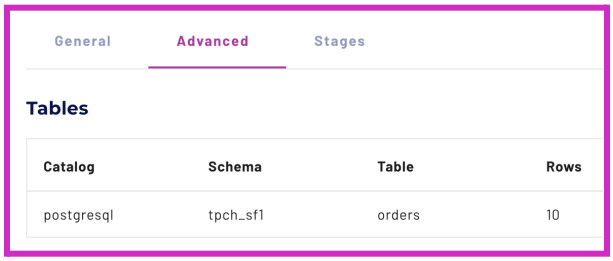
Step 3 - View query details

Now it's time to view the query details. To do this, click **View** icon on the results status bar as indicated in the image below.

Query details are displayed on the following page with the **General** tab selected. This will include your last executed SQL statement in the **Query text** section.



Click the **Advanced** tab to view more details. Review the **Tables** section to verify that `postgresql.tpch_sf1.orders` was read for this query.



When you have finished viewing query details, click on your browser's back button to return to precisely where you were in the **Query editor**.

Step 4 - Query the orders table using projection

Now it's time to query the Orders table. But you don't need all of the results, only specific columns.

You can use projection to narrow down your results. This allows you to select specific columns from a table. Insert the code below into the **Query editor**.

```
SELECT
 orderkey,
 custkey,
 totalprice
FROM
 postgresql.tpch_sf1.orders
WHERE
 orderkey < 100;
```


Step 5 - Query customer table

Next, it's time to look up the related `customer` table which is located in the `tpch` catalog.

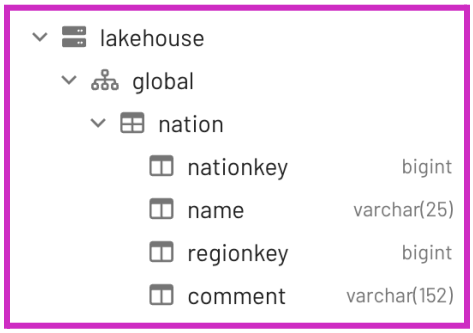
```
SELECT
 custkey,
 nationkey
FROM
 tpch.sf1.customer;
```

Verify from the **View** icon > **Advanced** > **Tables** that only the `tpch.sf1.customer` table was accessed.

Step 6 - Execute federated query joining two tables

Now it's time to write a query that's capable of looking across two tables from different catalogs. This is known as a federated query.

Begin by locating the `nation` table[1] in the `lakehouse` catalog's `global` schema.

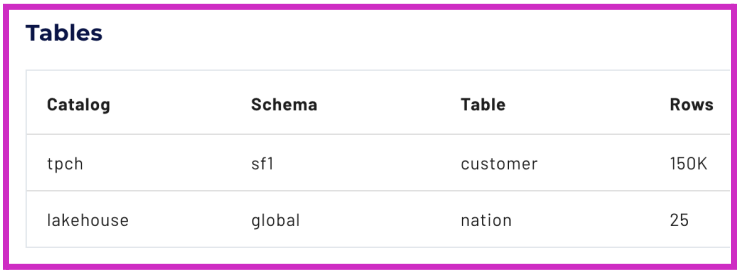


Now, run a *federated* query that joins the `tpch.sf1.customer` and `lakehouse.global.nation` tables to provide the full name of the country that is associated with each customer.

[1] This is an exact copy of the data from `tpch.sf1.nation`.

```
SELECT
  c.custkey,
  c.nationkey,
  n.name
FROM
  tpch.sf1.customer c
  JOIN lakehouse.global.nation n ON c.nationkey = n.nationkey
ORDER BY
  c.custkey;
```

Verify in your **Query details** that this query is *federating* content from two different catalogs which use different underlying data sources.



Step 7 - Federate across three tables

Enhance your query by joining a third table, `postgresql.tpch_sf1.orders`. Group the results by nation to find out which nations have the most customers.

```
SELECT
 n.name,
 count() AS nbr_custs
FROM
 postgresql.tpch_sf1.orders o
 JOIN tpch.sf1.customer c ON o.custkey = c.custkey
 JOIN lakehouse.global.nation n ON c.nationkey = n.nationkey
GROUP BY
 n.name
ORDER BY
 nbr_custs DESC;
```

Verify from **Query details** that this query is *federating* content from three different catalogs which are using different underlying data source connectors.

Tables

Catalog	Schema	Table	Rows
postgresql	tpch_sf1	orders	1.5M
tpch	sf1	customer	150K
lakehouse	global	nation	25

END OF LAB EXERCISE

Data lake tables

Lab 1: Create a schema and tables

Estimated completion time

- 20 minutes

Learning objectives

- This lab will show you how to perform basic database operations with Starburst Galaxy using Hive. This includes the creation of schemas and tables, using a number of different methods. The lab also introduces the concept of Inserts, record updates, and external tables.

Prerequisites

- Student setup - Lab 1: Create student account

Activities

1. Sign in and verify cluster
2. Create a schema
3. Create a table using `CREATE TABLE AS`
4. Verify match between data using the `EXCEPT` command
5. Generate Data Definition Language (DDL)
6. Connect catalog
7. CTAS region table from `tpch.sf1.region`
8. Create `friends` table
9. Verify file format using `SHOW CREATE TABLE`
10. Add a record to the table using `INSERT INTO`
11. Update a record
12. Join `friends` and `nation` tables
13. Using external tables in Hive

Step 1 - Sign in and verify cluster

Sign in and verify the `students` role is selected in the upper-right corner of the screen.

Navigate to the **Clusters** list and take any corrective action required to ensure `aws-us-east-1-free` is reporting a **Status** of `Running` before continuing.

Navigate to **Query** > **Query editor** and expand `aws-us-east-1-free` under the cluster list.

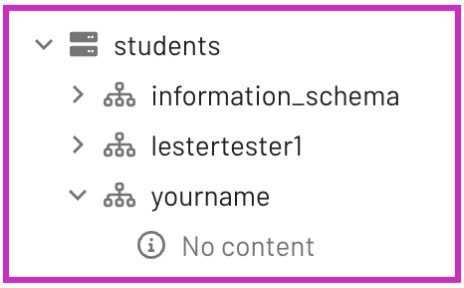
Step 2 - Create a schema

In the editor pane, create a new schema with the following command. Make sure that you replace `yourname` with your actual name, or another identifier that you prefer. **Only use lowercase characters and numbers; no special characters or spaces.**

```
CREATE SCHEMA
 students.yourname;
```

Confirm that the schema named `students.yourname` that you just created is visible. Expand the `students` catalog. Confirm that your new schema is empty, as pictured in the image below. There will be other schemas listed.

> **Note:** *Remember, if you named your schema as `yourname`, create the schema again with the prerequisite lab's instructions.*



Run the following SQL command to verify your newly created schema is empty. This provides secondary confirmation that there is nothing in it.

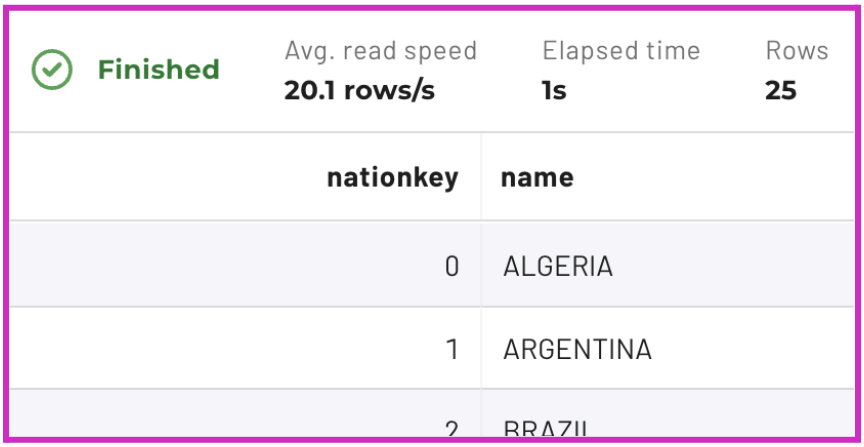
```
SHOW TABLES
FROM
 students.yourname;
```

Step 3 - Create a table using `CREATE TABLE AS`

Next, it's time to create a new table within your student schema following the format and contents of the `tpch.sf1.nation` table.

To do this, start by running a query against the `tpch.sf1.nation` table to verify there are 25 rows present. This helps us to ensure that we understand what information is held in the table before we recreate the same information in the student schema.

```
USE students.yourname;
SELECT
 *
FROM
 tpch.sf1.nation;
```

Next, review the CREATE TABLE AS (CTAS) documentation then create a table named `nation` in your schema that has the same format and content as `tpch.sf1.nation`.

```
CREATE TABLE
 nation AS
SELECT
 *
FROM
 tpch.sf1.nation;
```

Verify in the schema hierarchy that your new table exists and has the correct structure and content. Now use the `DESCRIBE` command to achieve the same output using SQL.

```
DESCRIBE nation;
```

Step 4 - Verify match between data using the `EXCEPT` command

After running `SELECT` queries on the original table and your `nation` table, run the EXCEPT query listed below. It will verify that the two tables contain the same data.

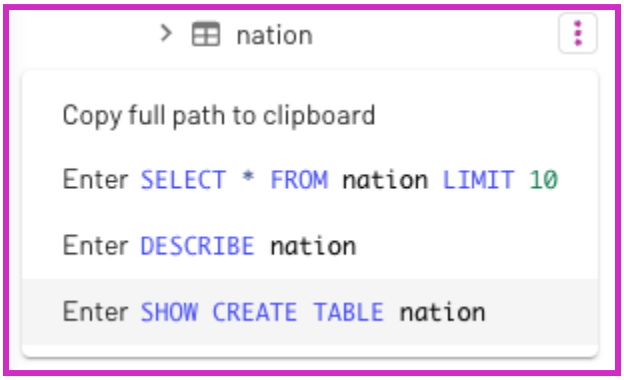
```
SELECT * FROM nation
EXCEPT
SELECT * FROM tpch.sf1.nation;
```

Step 5 - Generate Data Definition Language (DDL)

When you create a table, you need to define the data structure that will be employed. This is known as the Data Definition Language (DDL). Starburst helps you automate this process.

In the left-hand navigation bar, click the vertical ellipses menu under the `students.yourname.nation` table to generate the Data Definition Language (DDL) for that table.

Select the `SHOW CREATE TABLE` option shown in the image below.



The system generates the following output which has two focus areas.

Output

```
CREATE TABLE students.yourname.nation (
   nationkey bigint,
   name varchar(25),
   regionkey bigint,
   comment varchar(152)
)
WITH (
   format = 'ORC',
   type = 'HIVE'
)
```

The first focus area, noted blue, defines the logical representation of the table. It provides information about the schema which will be applied to the table and is similar to the approach used to construct a table in other database systems.

The second focus area, noted in red, defines two other crucial aspects of schema-on-read solutions, specifically:

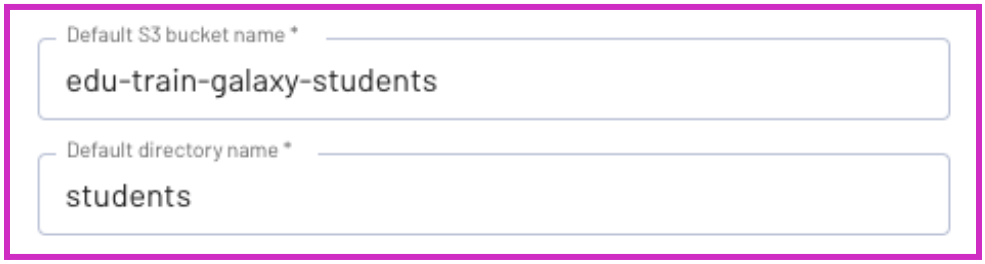
- The file format being used, in this case, ORC.
- The table format used, in this case, Hive.

Step 6 - Locate the data

Notice that you did not define a specific folder location in the `CREATE TABLE` statement in the last step. Although a specific location can be defined if needed, it can also be left undefined. When left to the default, it is based on where the schema itself is rooted in the data lake hierarchy.

Note: You do not have access to view the configuration of catalogs in the UI, nor to view S3 via the AWS management console. Screenshots are presented to aid in your understanding of where the data for the `nation` table is located.

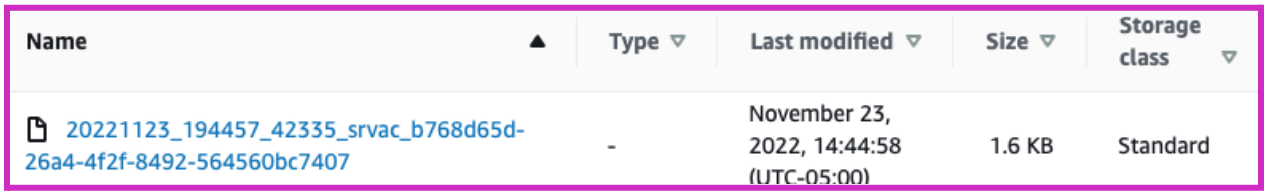
When the instructor created the `students` catalog, the following elements were visible.



By default, the schema is stored in a folder name below the **Default directory name**. Notice `students` directory name and the bucket named `edu-train-galaxy-students`. The table name is then another folder directly below the schema's folder.

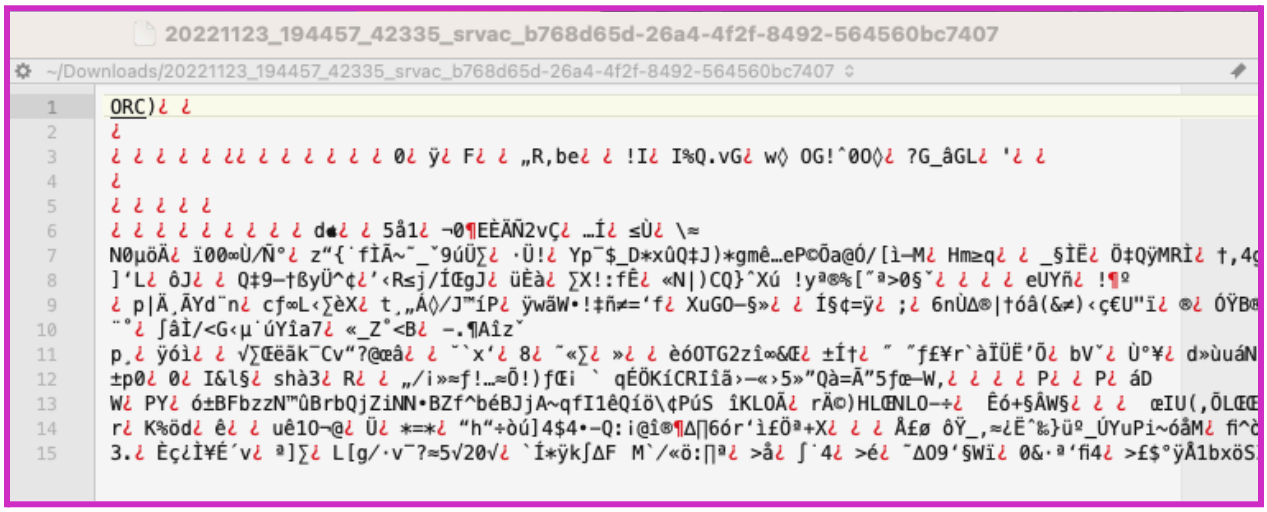
This means your table's base directory is `students/yourname/nation`. If you had access to AWS S3, you would see this directory tree.

The folder contains data files. In this case, you only have a single file because the table only has 25 rows.



As you can see below, the contents of this ORC file are not in a humanly readable format and cannot be read using a text editor.

Note: *The ORC tools jar is a straightforward way to visualize the data within. File format options will be discussed further in the next lesson.*



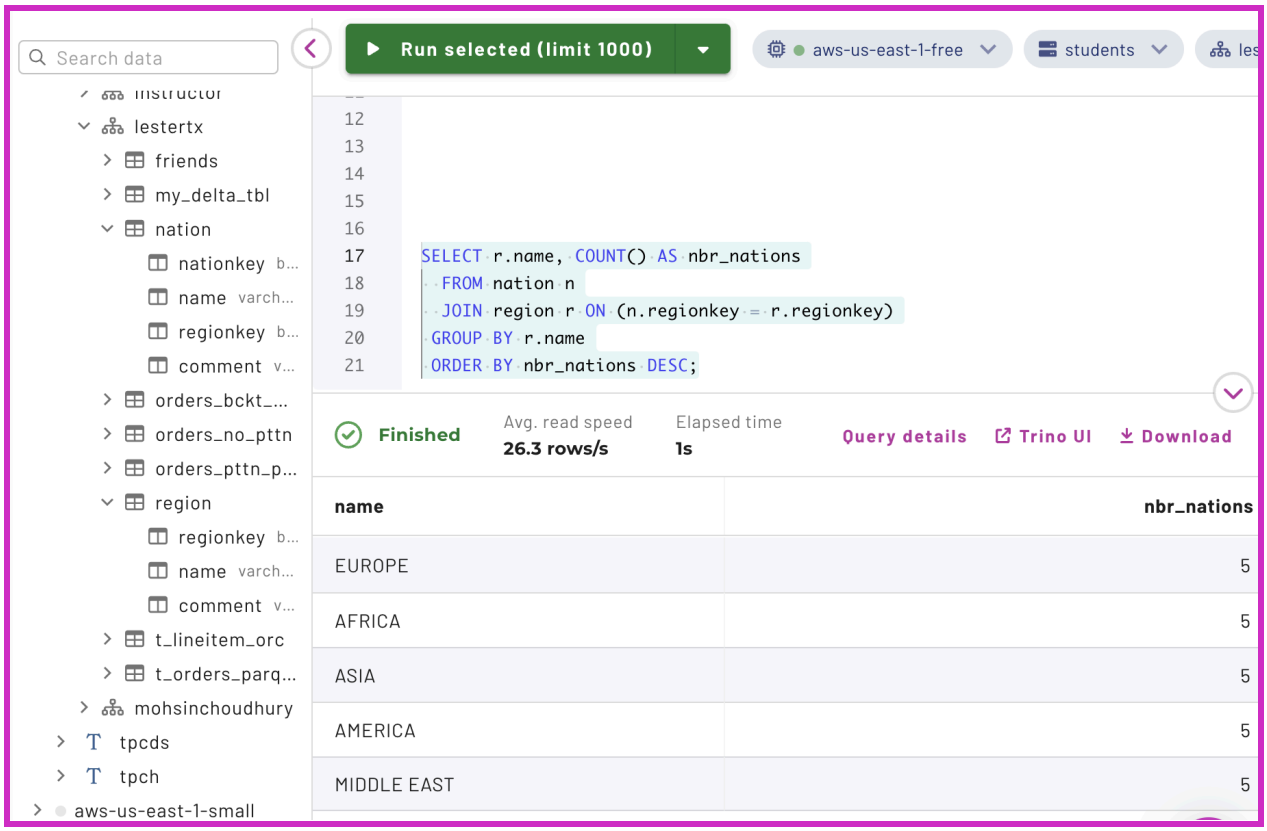
Step 7 - CTAS region table from `tpch.sf1.region`

Now it's time to use a similar process to copy the data from the `tpch.sf1.region` table into a new `region` table in your schema. Write another CTAS command using the code below.

```
CREATE TABLE
 region AS
SELECT
 *
FROM
 tpch.sf1.region;
```

Now join the two tables to determine the number of nations in each region.

```
SELECT r.name, COUNT() AS nbr_nations
 FROM nation n
 JOIN region r ON (n.regionkey = r.regionkey)
GROUP BY r.name
ORDER BY nbr_nations DESC;
```

Step 8 - Create `friends` table

Next, it's time to create a new table that will contain information about your friends. Use the code below.

```
CREATE TABLE friends (
   name varchar(100),
   year_met smallint,
   fav_color varchar(50),
   nation_born bigint
);
```

Notice that you did not define the `WITH` section. The location of the file system will default to something similar to `//filesystem/catalog/schema/table`. The folder will not be created until the table is populated with data.

Step 9 - Verify file format using `SHOW CREATE TABLE`

If the file format is not identified, the table will use the catalog's default file format. Verify that the file format is ORC by running a `SHOW CREATE TABLE` statement.

```
SHOW CREATE TABLE friends;
```


Your output should look like the following.

Output

```
CREATE TABLE students.yourname.friends (
   name varchar(100),
   year_met smallint,
   fav_color varchar(50),
   nation_born bigint
)
WITH (
   format = 'ORC',
   type = 'HIVE'
)
```

As for the `type` property above, you will learn more about "table formats" in an upcoming lesson.

Step 10 - Add a record to the table using `INSERT INTO`

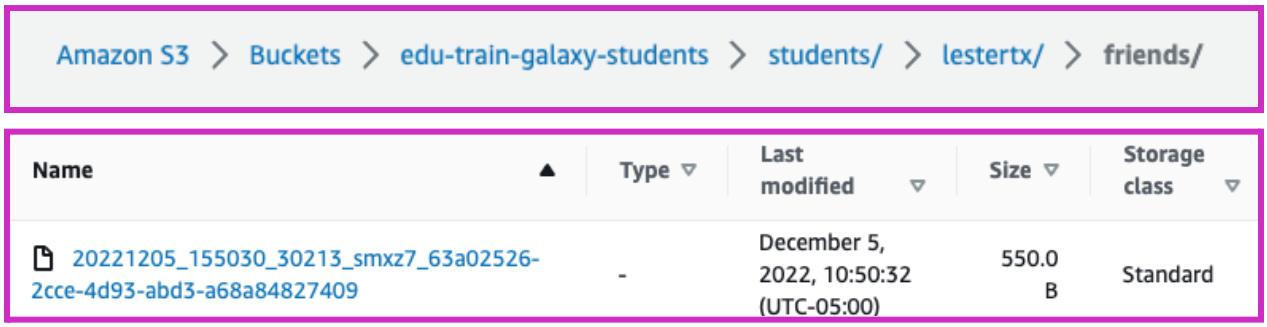
Now that the table is created, you can add records individually to it using the `INSERT INTO` command.

Add a record to this table using the following data.

```
INSERT INTO
 friends (name, year_met, fav_color, nation_born)
VALUES
 ('Fred', 2002, 'brown', 13);
```

When you add a record, it's considered best practice to check that the record has been successfully added. To do this, run a query to make sure the record was added to the table as intended. You should have all of the information you need to run this, but if you get stuck, review the previous steps or ask the instructor for help.

As before, you do not have access to view the S3 UI directly. These screenshots will help you understand what is present. The S3 object store now contains a folder for this table and a single file for this record. It should look similar to the image below.



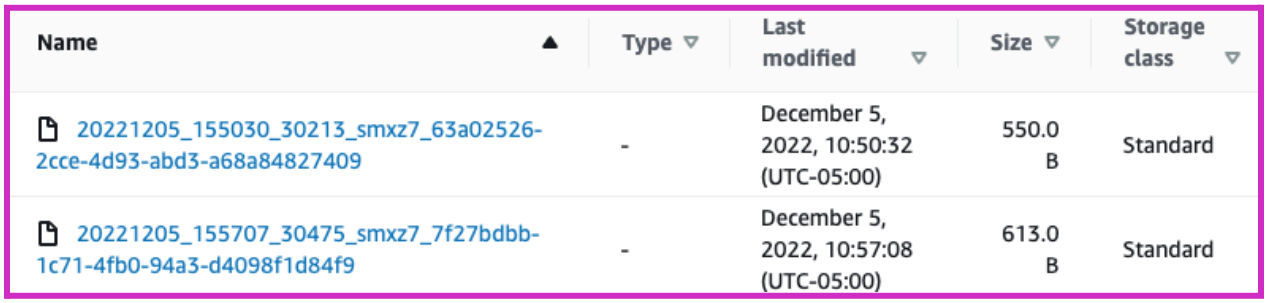
Now that you've added one record, try adding three records in the same `INSERT INTO` command.

```
INSERT INTO
 friends (name, year_met, fav_color, nation_born)
VALUES
 ('Susie', 2012, 'blue', 9),
 ('Bertha', 1999, 'pink', 24),
 ('Franklin', 2019, 'green', 24);
```

Verify the records have been added by running a query. There should now be four rows in the table.

```
SELECT * FROM friends;
```

If you could see the table's S3 folder you would notice there are two files now similar to the following example.



One of the files was created with the original `INSERT` statement and the other from the second `INSERT`. The three recently inserted records are all located in the second file. In the next lab, you will explore these files in more detail using the Hive special columns feature.

Run the original `INSERT` statement again.

```
INSERT INTO
 friends (name, year_met, fav_color, nation_born)
VALUES
 ('Fred', 2002, 'brown', 13);
```

Do you see the same record twice?

Finished | Avg. read speed 5.2 rows/s | Elapsed time 0.96s | Rows 5 | Query details

name	year_met	fav_color	nation_born
Susie	2012	blue	9
Bertha	1999	pink	24
Franklin	2019	green	24
Fred	2002	brown	13
Fred	2002	brown	13

The image above shows that one of your records has been duplicated. This happened because you did not declare a primary key (PK). It highlights a key point: when working with big data schema-on-read tables such as this, you will not have the chance to create (or validate) a primary key. Instead, data integrity activities like this are managed by the process that populates the table. This will be covered in greater detail in the data pipelines lesson.

Step 11 - Update a record

Often, it is necessary to update a record. ***Try*** to update one of the records you already created using the code below.

```
UPDATE friends
   SET fav_color = 'purple'
 WHERE name = 'Susie';
```

This raised the following error message.

Row-level modifications are not supported for Hive tables

The error message identifies the problem. Do you think there will be a problem with trying a `DELETE` on this same row? Try it and see.

```
DELETE FROM friends
 WHERE name = 'Susie';
```

This next error message reads as follows.

> Modifying Hive table rows is constrained to deletes of whole partitions

You will address partitions in a future module, but Hive tables are generally insert-only in nature. As mentioned previously, alternate "table formats" which allow full `INSERT`, `UPDATE`, and `DELETE` features, as well as usage of the `MERGE` command, are covered in a future lesson.

Additional efforts must be employed with Hive tables to recreate the underlying files from which the table gets queried. Some of these approaches are covered in the data pipelines lesson.

Step 12 - Join `friends` and `nation` tables

Join the `friends` table with the `nation` table you created earlier to determine how many of your friends were born in each nation.

```
SELECT n.name, COUNT() AS nbr_friends
  FROM friends f
  JOIN nation n ON (f.nation_born = n.nationkey)
 GROUP BY n.name
 ORDER BY nbr_friends DESC;
```

name	nbr_friends
UNITED STATES	2
JORDAN	2
INDONESIA	1

Adjust your query to join your `friends` on another schema's (from the `students` catalog) `nation` table, such as `students.ANOTHERNAME.nation`.

Note: If you cannot find a `nation` table from anyone else's schema, you can always use the `tcph.tiny` schema.

With traditional RDBMS software, you can enforce the relationship between these two tables with a foreign key (FK). As with PK constraints, Hive tables do not offer this type of integrity check. This allows you to add a record with a value that is not in the `nation` table, as illustrated in the following statement.

```
INSERT INTO
 friends (name, year_met, fav_color, nation_born)
VALUES
 ('John', 2021, 'khaki', 999);
```

Verify that this is the only record that does not have a matching `nation` table entry.

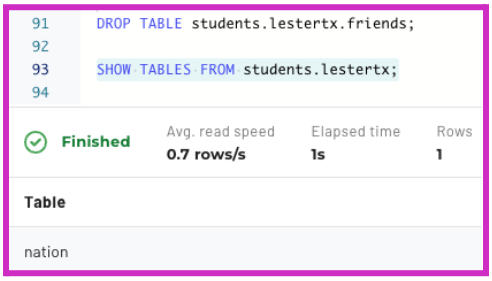
```
SELECT *
  FROM friends
 WHERE nation_born NOT IN (
     SELECT nationkey FROM nation
 );
```

name	year_met	fav_color	nation_born
John	2021	khaki	999

Similar to the PK, the burden of ensuring a logical FK constraint is **not** introduced into the data to be queried is the responsibility of the data pipeline process.

Eliminate the `friends` table now that you are finished with it.

```
DROP TABLE friends;
```



Step 13 - Using external tables in Hive

Up to this point, you have been creating Hive tables that are sometimes called **managed tables**. Generally speaking, you let the system identify where the underlying location will be, as the earlier example explained. While each table is configured to a specific directory and has a known file type, you do not typically place files on the filesystem directly. The Trino engine does the heavy lifting for us as a result of a command such as `INSERT`.

Additionally, when a `DROP TABLE` command is issued on a managed table, the underlying data is deleted along with the table's metadata.

External tables are the other primary type of Hive tables. External tables are often located in an alternate data lake location than where Trino would typically store them. You can identify these tables by the presence of the `external_location` property in the DDL. Review and execute the query below.

```
CREATE TABLE
 external_logs (
   event_time TIMESTAMP,
   ip_address VARCHAR (15),
   app_name VARCHAR (25),
   process_id SMALLINT,
   log_type VARCHAR (15),
   log_level VARCHAR (15),
   message_id VARCHAR (15),
   message_details VARCHAR (555)
 )
WITH
 (
   external_location =
's3://edu-train-galaxy/serverlogs/logs_5min_ingest_csv/',
   format = 'TEXTFILE',
   textfile_field_separator = ','
 );
```

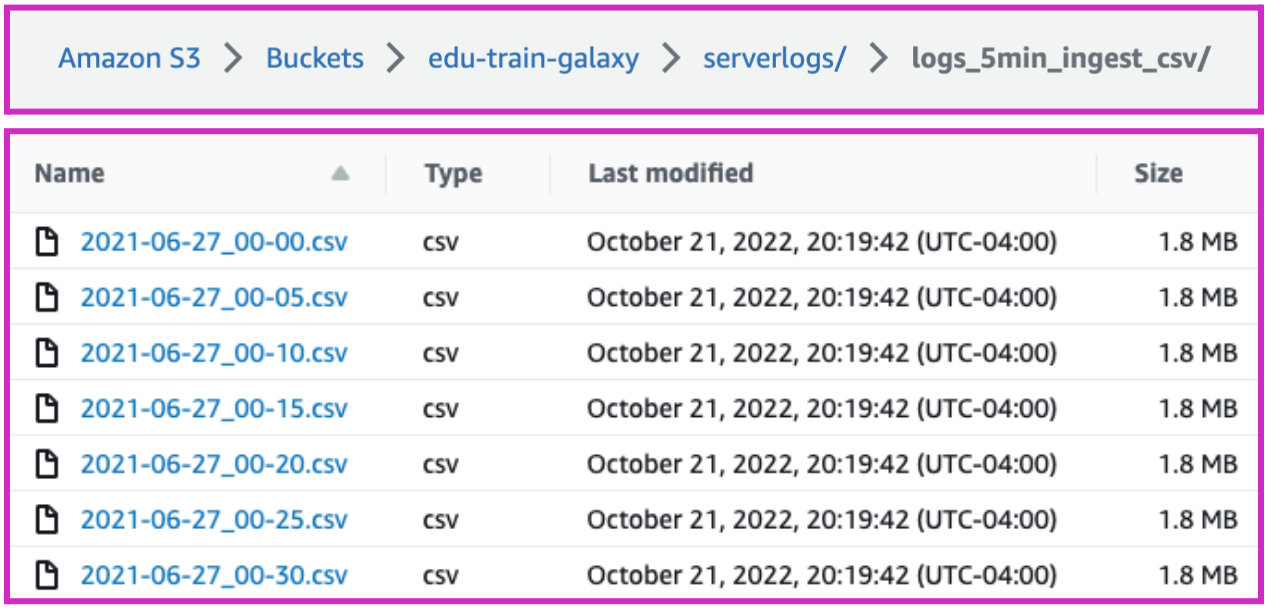
Verify the results from a `SELECT` statement against this new table look appropriate.

```
SELECT * FROM external_logs;
```

External tables are often utilized when another system is creating the files. For example, a frequent scenario is an external table being used for recently ingested files that are being loaded into the data lake. In addition to allowing other systems to populate data into the table, these data lake files are easily consumable from other frameworks and systems.

In the ingestion model, you often see similar-sized files placed in the table's underlying directory periodically. Starburst does not care what these files are named, but depending on the system saving the files, you may get some heuristic information from the file names. You might be able to decipher the pattern from these example files being used in the table you just created.

Remember, you do not have access to S3 directly and these screenshots are presented to aid in your understanding.



Note: *you will see these files again in the next lab as the basis of some of your performance enhancement techniques.*

Another huge difference with external tables is related to when the table is dropped. Only the metadata is removed. The actual data is left alone. To validate this, drop the `external_logs` table you just created. Recreate it and confirm the data is still available.

END OF LAB EXERCISE

Lab 2: Investigate Hive's special columns

Estimated completion time

- 15 minutes

Learning objectives

- This lab will explore Hive's use of columns and special columns. The goal is to show the kind of metadata captured in special columns, how you can access that data, and some of the formatting that you can apply to the output from these columns. You will then create a test table to simulate testing in real-world scenarios.

Prerequisites

- Student setup - Lab 1: Create student account

Activities

1. Prepare for the exercise
2. Understanding special columns in Hive
3. Show location of table data
4. Return special column values
5. Remove duplicate results using `DISTINCT` command
6. Locate results file
7. Use common table expression and string functions
8. Create table to test hidden columns
9. Improve clarity of results
10. Add more records

Step 1 - Prepare for the exercise

- Sign in and verify the `students` role is selected in the upper-right corner.
- Ensure the `aws-us-east-1-free` **Cluster** is reporting a **Status** of `Running`.
- In the **Query editor**, select `aws-us-east-1-free` in the cluster pulldown.

Note: *If you did not previously create a schema*, execute the following SQL statement. Make sure that you replace `yourname` with your actual name, or another identifier that you prefer. **Only use lowercase characters and numbers; no special characters or spaces.**

```
CREATE SCHEMA students.yourname;
```

Step 2 - Understanding special columns in Hive

This lab will explore the special columns functionality available for Hive tables.

In addition to defined columns, the Hive connector automatically exposes the metadata for several hidden columns in each table. These include:

- `$bucket`: Bucket number for this row
- `$path`: Full file system path name of the file for this row
- `$file_modified_time`: Date and time of last modification of the file for this row
- `$file_size`: Size of the file for this row
- `$partition`: Partition name for this row

This lab will focus on the `$path` and `$file_size` hidden columns. Additional hidden columns will be explored in a later exercise.

Step 3 - Show location of table data

Navigate to **Query** > **Query editor** and expand `aws-us-east-1-free` under the list of clusters.

Run the following command to see the location of the table's data.

```
SHOW CREATE TABLE lakehouse.global.nation;
```

Unfortunately, the output does not clarify where this data is located.

Output

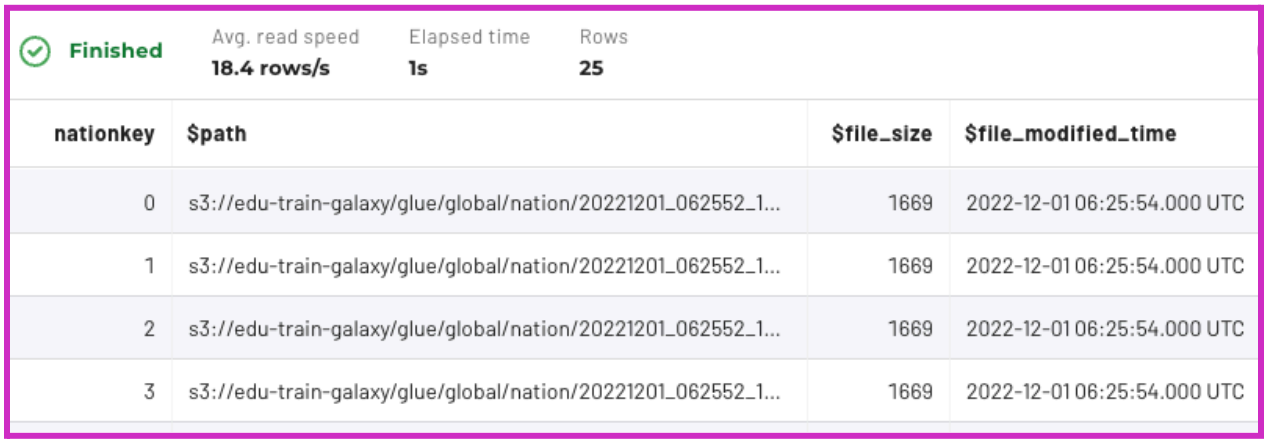
```
CREATE TABLE lakehouse.global.nation (
   nationkey bigint,
   name varchar(25),
   regionkey bigint,
   comment varchar(152)
)
WITH (
   format = 'ORC',
   type = 'HIVE'
)
```

Step 4 - Return special column values

There are only 25 rows in this table. Let's find the values of a few of these special columns.

The following code returns the `$path`, `$file_size`, and `$file_modified_time` values, as pictured in the image below.

```
SELECT
 nationkey,
 "$path",
 "$file_size",
 "$file_modified_time"
FROM
 lakehouse.global.nation;
```



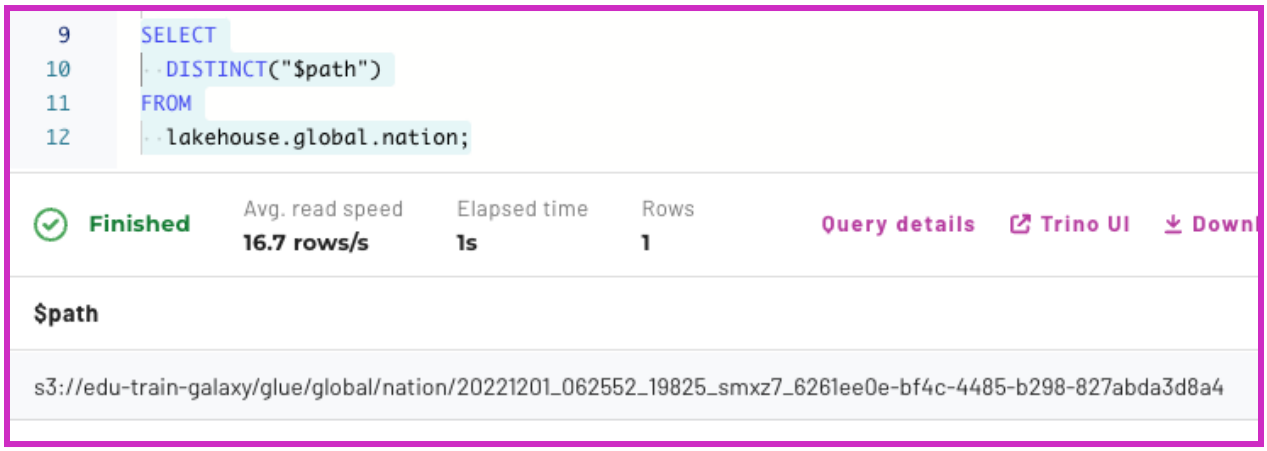
Finished | Avg. read speed: 18.4 rows/s | Elapsed time: 1s | Rows: 25

nationkey	$path	$file_size	$file_modified_time
0	s3://edu-train-galaxy/glue/global/nation/20221201_062552_1...	1669	2022-12-01 06:25:54.000 UTC
1	s3://edu-train-galaxy/glue/global/nation/20221201_062552_1...	1669	2022-12-01 06:25:54.000 UTC
2	s3://edu-train-galaxy/glue/global/nation/20221201_062552_1...	1669	2022-12-01 06:25:54.000 UTC
3	s3://edu-train-galaxy/glue/global/nation/20221201_062552_1...	1669	2022-12-01 06:25:54.000 UTC

Step 5 - Remove duplicate results using `DISTINCT` command

The query results pane makes it hard to read the value(s) from the `$path` hidden column. To fix this, you can remove duplicates and limit the results to show only the column's value(s) using the `DISTINCT` command. Try it now and see the difference.

```
SELECT DISTINCT("$path")
  FROM lakehouse.global.nation;
```



```
9   SELECT
10    DISTINCT("$path")
11  FROM
12    lakehouse.global.nation;
```

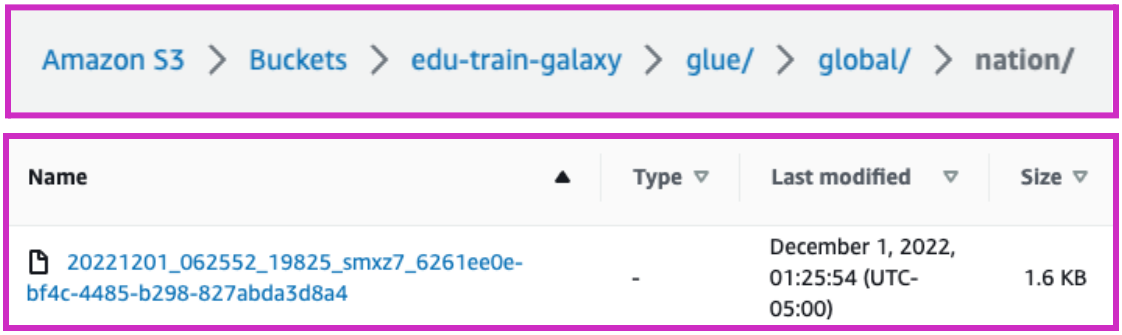
Finished | Avg. read speed: 16.7 rows/s | Elapsed time: 1s | Rows: 1 | Query details | Trino UI | Downl

$path
s3://edu-train-galaxy/glue/global/nation/20221201_062552_19825_smxz7_6261ee0e-bf4c-4485-b298-827abda3d8a4

Step 6 - Locate the results file

The results are output to a single file held in an S3 bucket. In this example, the file is located in the `s3://edu-train-galaxy/glue/global/nation/` directory.

The images below are from the instructor's AWS console showing an example of the location and the files list.



Step 7 - Common table expression and string functions

You can use a common table expression (CTE) and string functions to display the folder location. This makes the path easier to view.

Use the following query to return the path down to the directory that contains the file. The result should resemble the image below.

Note: This SQL is presented as an aid to review the underlying contents on the data lake and there is no need to focus on interpreting it at this time.

```
WITH
 distinct_paths AS (
   SELECT DISTINCT
     ("$path") AS full_path
   FROM
     lakehouse.global.nation
 )
SELECT
 REVERSE(
   SUBSTRING(
     REVERSE(full_path),
     POSITION('/' IN REVERSE(full_path))
   )
 ) AS dir_name
FROM
 distinct_paths;
```

dir_name
s3://edu-train-galaxy/glue/global/nation/

Step 8 - Create table to test hidden columns

Now it's time to test the hidden columns. To do this, you will need to create a new table using the code below.

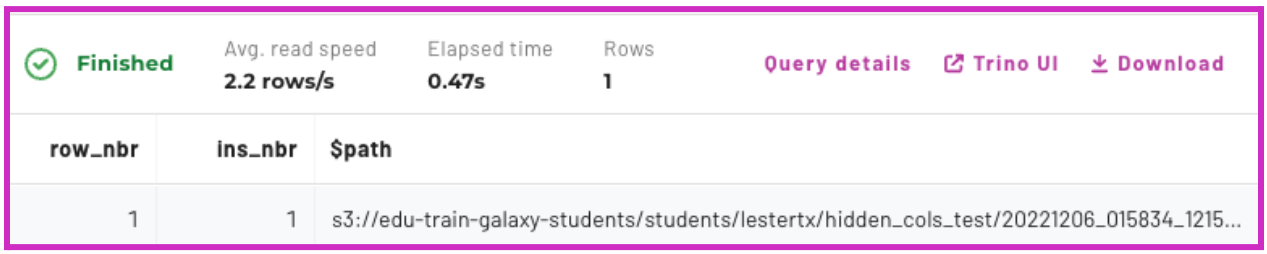
```
USE students.yourname;


CREATE TABLE hidden_cols_test (
   row_nbr int, ins_nbr int
);
```

Now add one record using the `INSERT` statement. This will create the first file (holding this single record) in the table's data lake folder.

```
INSERT INTO
 hidden_cols_test (row_nbr, ins_nbr)
VALUES
 (1, 1);
```

Now it's time to verify that the single file exists in the `hidden_cols_test` table using the code below.

```
SELECT
 row_nbr, ins_nbr, "$path"
FROM
 hidden_cols_test
ORDER BY
 row_nbr, ins_nbr;
```



Step 9 - Improve clarity of results

The clarity of the query results displaying this hidden column could still be improved further. Assume that you want to show just the file name. The statement below is an enhancement of the one presented in **Step 7** and will display a file name only, without the directory listed.

Note: As before, this SQL is presented as an aid to review the underlying contents on the data lake and there is no need to focus on interpreting it at this time.

```
--NOTE: Convenience SQL (NO NEED TO REVIEW THIS!!)
WITH
all_rows_file_names AS (
  SELECT row_nbr, ins_nbr AS file_nbr,
         ("$path") AS full_path
    FROM hidden_cols_test
)
SELECT
  row_nbr, file_nbr,
  SPLIT_PART(full_path, '/', 7) AS file_name
  FROM all_rows_file_names
 ORDER BY row_nbr;
```

row_nbr	file_nbr	file_name
1	1	20230511_224847_81659_c9yau_0909ab6a-afac-4d6a-8ea2-32fc8cb7ce0a

The output above shows us that the first `INSERT` statement (of a single row) was recorded in a single file.

Step 10 - Add more records

Now it's time to add results and view the impact on the test tables. Add two more records with a second `INSERT` statement.

```
INSERT INTO
 hidden_cols_test (row_nbr, ins_nbr)
VALUES
 (2, 2),
 (3, 2);
```

Run the **Step 9** query again to verify that this second `INSERT` statement (of two more rows) wrote another file containing both values.

row_nbr	file_nbr	dir_name
1	1	20230406_232600_03236_tsz8n_1f511a03-511f-4ffd-aabd-2e838cb1a34c
2	2	20230406_232855_04519_tsz8n_ca695869-fc74-4179-b0f1-72bee5614eb8
3	2	20230406_232855_04519_tsz8n_ca695869-fc74-4179-b0f1-72bee5614eb8

Now add three more records with a third `INSERT` statement.

```
INSERT INTO
 hidden_cols_test (row_nbr, ins_nbr)
VALUES
 (4, 3),
 (5, 3),
 (6, 3);
```

Running the **Step 9** query one last time verifies this third `INSERT` statement (of three rows) wrote one final file with all three values.

row_nbr	file_nbr	dir_name
1	1	20230406_232600_03236_tsz8n_1f511a03-511f-4ffd-aabd-2e838cb1a34c
2	2	20230406_232855_04519_tsz8n_ca695869-fc74-4179-b0f1-72bee5614eb8
3	2	20230406_232855_04519_tsz8n_ca695869-fc74-4179-b0f1-72bee5614eb8
4	3	20230406_232911_04557_tsz8n_236f6b83-8cdb-4c30-9d79-3daffbf74897
5	3	20230406_232911_04557_tsz8n_236f6b83-8cdb-4c30-9d79-3daffbf74897
6	3	20230406_232911_04557_tsz8n_236f6b83-8cdb-4c30-9d79-3daffbf74897

You cannot view S3, but here is a view from the instructor's AWS console. You can see that the three distinct file names above are exactly how they are stored in S3.

END OF LAB EXERCISE

Data lake performance

Lab 1: Create tables with multiple file formats

Estimated completion time

- 20 minutes

Learning objectives

- In this lab, you will gain experience creating tables in different file formats using the CTAS method. You will also join these tables, using a variety of techniques to build and consolidate your skills.

Prerequisites

- Student setup - Lab 1: Create student account

Activities

1. Prepare for the exercise
2. Create managed parquet table for orders
3. Review summary information
4. Create managed ORC table for line items
5. Join tables
6. Create managed Avro table for customer
7. Verify `nation` table
8. Modify join query to include new table
9. Create managed RCText table for region
10. Create managed JSON table for supplier

Step 1 - Prepare for the exercise

- Sign in and verify the `students` role is selected in the upper-right corner.
- Ensure the `aws-us-east-1-free` **Cluster** is reporting a **Status** of `Running`.
- In the **Query editor**, select `aws-us-east-1-free` in the cluster pulldown.

Note: *If you did not previously create a schema*, execute the following SQL statement. Make sure that you replace `yourname` with your actual name, or another identifier that you prefer. **Only use lowercase characters and numbers; no special characters or spaces.**

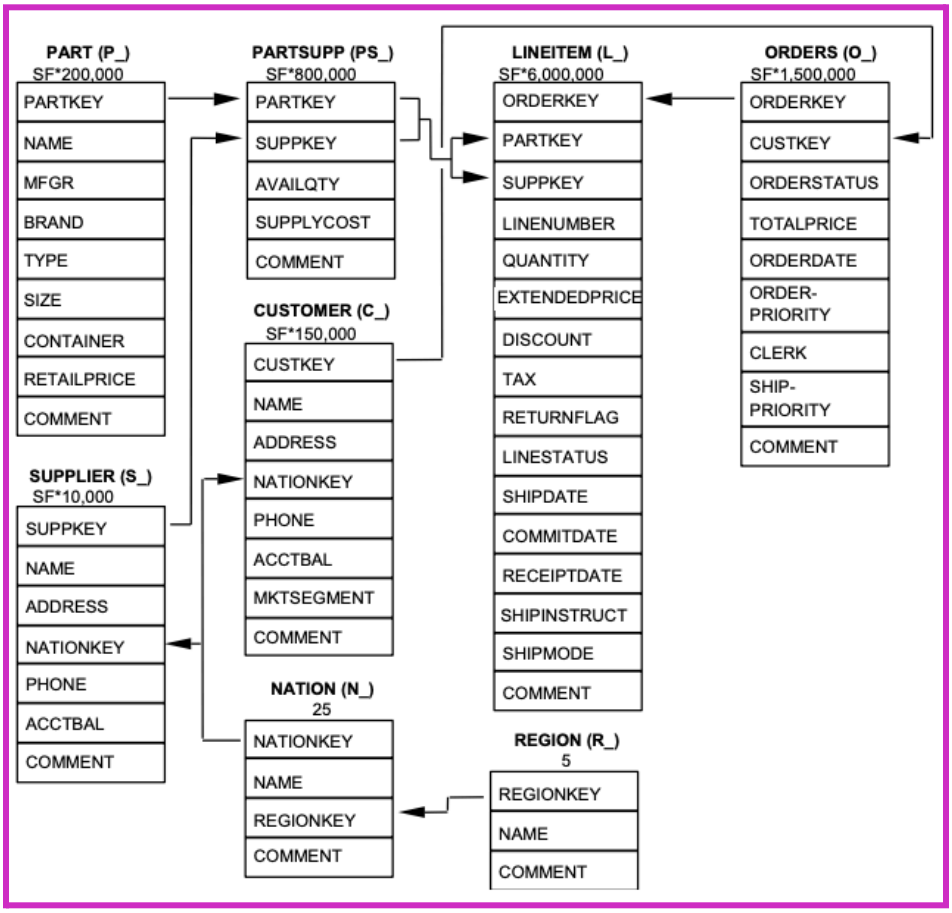
```
CREATE SCHEMA students.yourname;
```

Step 2 - Create managed parquet table for orders

In this exercise, you will create tables using a variety of supported file types and leverage them together in a multi-table join statement.

The tables that exist in `tpch.sf1` schema and the logical relationships between them are listed below.

Review these relationships thoroughly. You'll need to use these relationships to proceed further in the lab.



Using the CTAS statement below, create a managed table in your own schema that contains the contents of `tpch.sf1.orders`. Use the *Parquet* file format for this new table.

```
USE students.yourname;


CREATE TABLE
 t_orders_parquet
WITH
 (format = 'Parquet') AS
SELECT
 *
FROM
 tpch.sf1.orders;
```

Verify that the table was properly created.

```
SELECT
 *
FROM
 t_orders_parquet
LIMIT
 25;
```

Step 3 - Review summary information

It is considered best practice to review the S3 bucket where your data is stored to gain summary information about the newly created files.

Use the SQL query below to generate the summary.

```
SELECT format_number(AVG("$file_size")) AS avg_file_size,
     COUNT() AS tot_nbr_files,
     format_number(SUM("$file_size")) AS tot_file_size
FROM
  (SELECT DISTINCT "$path", "$file_size"
     FROM t_orders_parquet);
```

Step 4 - Create managed ORC table for line items

Using a CTAS statement, create a managed table in your own schema that contains the contents of `tpch.sf1.lineitem`. Use the *ORC* file format for this new table.

```
CREATE TABLE
 t_lineitem_orc
WITH
 (format = 'ORC') AS
SELECT
 *
FROM
 tpch.sf1.lineitem;
```

Verify that the table was properly created.

```
SELECT
 *
FROM
 t_lineitem_orc
LIMIT
 25;
```

Run a query to determine some summary information about the newly created files.

```
SELECT format_number(AVG("$file_size")) AS avg_file_size,
     COUNT() AS tot_nbr_files,
     format_number(SUM("$file_size")) AS tot_file_size
FROM
  (SELECT DISTINCT "$path", "$file_size"
     FROM t_lineitem_orc);
```

Step 5 - Join tables

Investigate these two new tables to determine which columns should be used to join them.

Run the query below.

```
SELECT
 o.orderkey,
 l.linenumber
FROM
 t_lineitem_orc AS l
 JOIN t_orders_parquet AS o ON l.orderkey = o.orderkey
WHERE
 l.orderkey = 1935460;
```

Step 6 - Create managed Avro table for customer

Using a CTAS statement, create a managed table in your schema that contains the contents of `tpch.sf1.customer`. Use `format = 'Avro'` for this new table.

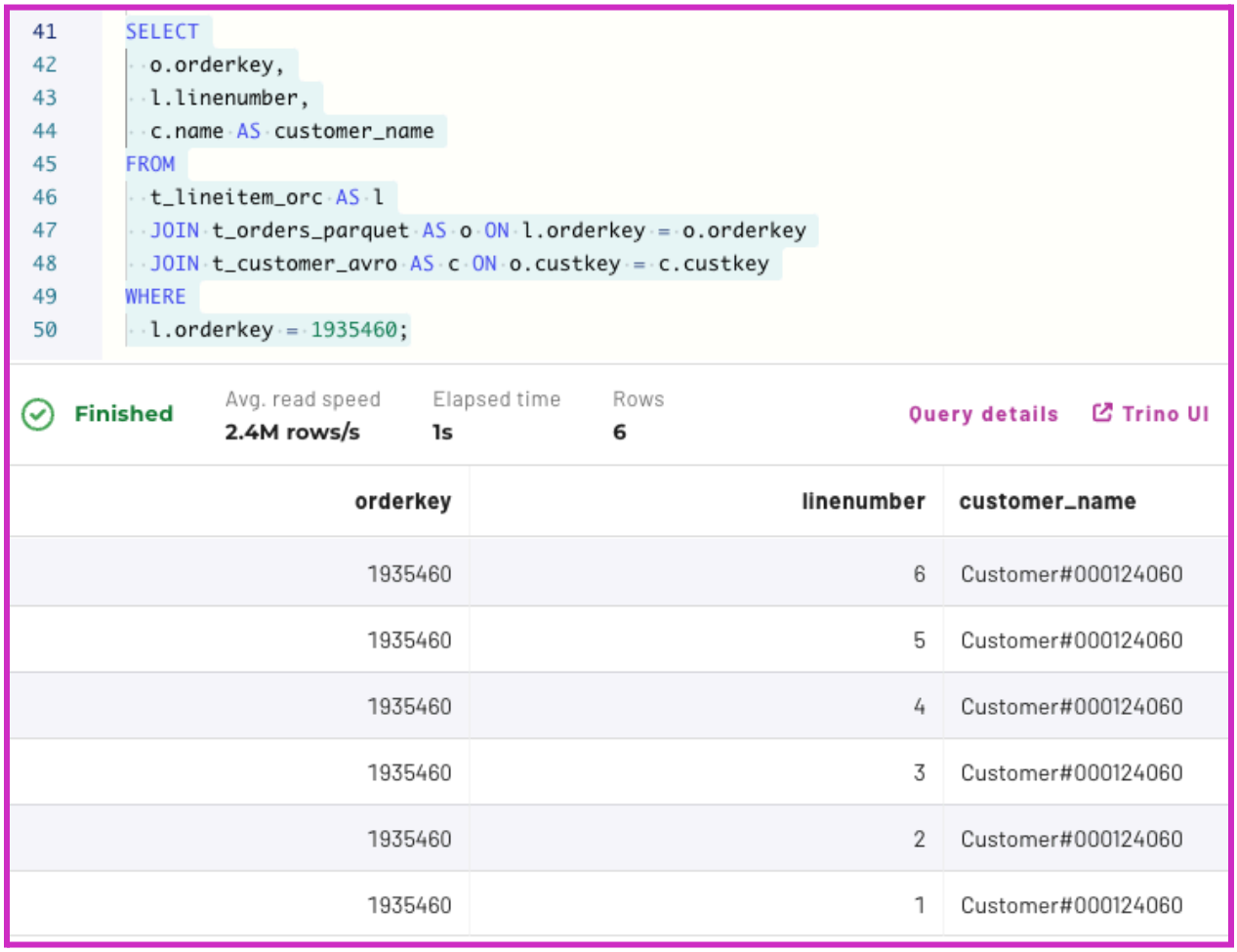
```
CREATE TABLE
 t_customer_avro
WITH
 (format = 'Avro') AS
SELECT
 *
FROM
 tpch.sf1.customer;
```

Verify that the table was created properly and then determine some summary information about the newly created files using the procedure described in **Step 3**.

Enhance the join query to include this new table and include its `name` column in the results.

```
SELECT
 o.orderkey,
 l.linenumber,
 c.name AS customer_name
FROM
 t_lineitem_orc AS l
 JOIN t_orders_parquet AS o ON l.orderkey = o.orderkey
 JOIN t_customer_avro AS c ON o.custkey = c.custkey
WHERE
 l.orderkey = 1935460;
```

The result should look similar to the image below.



```
41  SELECT
42    o.orderkey,
43    l.linenumber,
44    c.name AS customer_name
45  FROM
46    t_lineitem_orc AS l
47    JOIN t_orders_parquet AS o ON l.orderkey = o.orderkey
48    JOIN t_customer_avro AS c ON o.custkey = c.custkey
49  WHERE
50    l.orderkey = 1935460;
```

Finished | Avg. read speed 2.4M rows/s | Elapsed time 1s | Rows 6 | Query details | Trino UI

orderkey	linenumber	customer_name
1935460	6	Customer#000124060
1935460	5	Customer#000124060
1935460	4	Customer#000124060
1935460	3	Customer#000124060
1935460	2	Customer#000124060
1935460	1	Customer#000124060

Step 7 - Verify `nation` table

You should already have a `nation` table from an earlier lab. If not, create a copy of `tpch.sf1.nation` with a CTAS statement using the default file format. You should leave out the `format` property.

Step 8 - Modify join query to include new table

Enhance the join query earlier in this lab to include the newly created table. Make sure to reference the `name` column in the results.

Your query should be similar to the one below.

```
SELECT
 o.orderkey,
 l.linenumber,
 c.name AS customer_name,
 n.name AS nation_name
FROM
 t_lineitem_orc AS l
 JOIN t_orders_parquet AS o ON l.orderkey = o.orderkey
 JOIN t_customer_avro AS c ON o.custkey = c.custkey
 JOIN nation AS n ON c.nationkey = n.nationkey
WHERE
 l.orderkey = 1935460;
```

Step 9 - Create more tables with additional file formats

Run the following three CTAS statements utilizing some of the additional supported file formats.

Note: These file formats are not recommended for new efforts, but are supported for legacy systems that leverage some, or all, of them.

```
CREATE TABLE t_region_rctext
 WITH (format = 'RCText') AS
   SELECT * FROM tpch.sf1.region;


CREATE TABLE t_part_sf
 WITH (format = 'SequenceFile') AS
   SELECT * FROM tpch.sf1.part;


CREATE TABLE t_supplier_json
 WITH (format = 'JSON') AS
   SELECT * FROM tpch.sf1.supplier;
```

Step 10 - Join 7 tables of differing file formats

Finally, join all of the new tables created in this exercise.

```
SELECT
 O.orderkey, l.linenumber,
 c.name AS customer_name,
 n.name AS nation_name,
 r.name AS region_name,
 p.name AS part_name,
 s.name AS supplier_name
FROM
 t_lineitem_orc AS l
 JOIN t_orders_parquet AS o ON l.orderkey = o.orderkey
 JOIN t_customer_avro AS c ON o.custkey = c.custkey
 JOIN nation AS n ON c.nationkey = n.nationkey
 JOIN t_region_rctext AS r ON n.regionkey = r.regionkey
 JOIN t_part_sf AS p ON l.partkey = p.partkey
 JOIN t_supplier_json AS s ON l.suppkey = s.suppkey
WHERE
 l.orderkey = 1935460;
```

The solution above was useful to determine all the joins. A more analytical view of the data may require calculating how many line items exist for each supplier/brand name combination for each customer region. That could yield a modified aggregation query such as the following.

```
SELECT
 s.name AS supplier_name, p.brand AS brand_name,
 r.name AS region_name, --customer region, not supplier
 COUNT() AS nbr_line_items
FROM
 t_lineitem_orc AS l
 JOIN t_orders_parquet AS o ON l.orderkey = o.orderkey
 JOIN t_customer_avro AS c ON o.custkey = c.custkey
 JOIN nation AS n ON c.nationkey = n.nationkey
 JOIN t_region_rctext AS r ON n.regionkey = r.regionkey
 JOIN t_part_sf AS p ON l.partkey = p.partkey
 JOIN t_supplier_json AS s ON l.suppkey = s.suppkey
GROUP BY
 ROLLUP (s.name, p.brand, r.name)
ORDER BY
 S.name, p.brand, r.name;
```

END OF LAB EXERCISE

Lab 2: Using columnar file formats and eliminating small files

Estimated completion time

- 40 minutes

Learning objectives

- In this lab, you will compare performance of small CSV files with more ideally sized ORC files which require fewer calls back to the data lake. You will learn how file formats and file size affect performance by spending less CPU time reading smaller amounts of bytes from the data lake. Additional improvements will surface from more efficient use of parallelization from the inherent benefits of columnar file formats.

Prerequisites

- Student setup - Lab 1: Create student account

Activities

1. Prepare for the exercise
2. Understanding the testing data
3. Row-oriented file formats vs. columnar file format
4. Ensure cluster is ready and metadata is cached
5. See reduction in bytes read with column-oriented files
6. Locate metrics for amount of data exchanged
7. Projection benefits with row-oriented files
8. Projection benefits with column-oriented files
9. Locate metrics for number of splits read
10. Validate small files problems
11. Embedded column statistics for filtering
12. Embedded column statistics for aggregations

Step 1 - Prepare for the exercise

- Sign in and verify the `students` role is selected in the upper-right corner.
- Ensure the `aws-us-east-1-small` **Cluster** is reporting a **Status** of `Running`.
- In the **Query editor**, select `aws-us-east-1-small` in the cluster pulldown.

Note: *If you did not previously create a schema*, execute the following SQL statement. Make sure that you replace `yourname` with your actual name, or another identifier that you prefer. **Only use lowercase characters and numbers; no special characters or spaces.**

```
CREATE SCHEMA students.yourname;
```

Step 2 - Understanding the testing data

This lab will focus on some of the well-known improvements centered on file formats and sizes that are designed to increase performance and scalability.

As you work through this lab, imagine that you are responsible for analyzing application log data at your company.

Initially, you will explore the log data by querying a table that is linked to raw data as it arrives in the data lake. You will also query other tables leveraging additional performance strategies.

In all cases, the logical structure of the log data will be the same. The table below shows that definition of the structure and also includes example data and additional notes.

Field Name	Datatype	Example Data	Additional Notes
Event Timestamp	Timestamp	2021-07-04 13:01:01	Format is yyyy-MM-dd HH:mm:ss
IP Address	String	202.101.36.182	Static set of 990 values
Application Name	String	Payroll	Static set of 26 values
Process ID	Integer	2397	Random number from 99 - 5999
Log Type	String	THREAT	Randomly selected from this list with highest probability from left to right; EVENT, AUDIT, REQUEST, AVAILABILITY, THREAT
Log Level	String	WARN	Randomly selected from this list with highest probability from left to right; INFO, DEBUG, WARN, TRACE, ERROR, FATAL
Message ID	String	CCE-5059	Random value with format XXX-9999
Message Details	String	more bugs Tomcat ERP bug bugs buffer overflow App95 10aebd44-78dc-459a-a678-01e586bd6309	Random assembly of 7 words and/or phrases from a static set plus a UUID at the end

For this lab, the dataset was derived from a data generator. The generator created the log records in a delimited file format; in this case, comma-separated values, or CSV.

Below is a sample of records from one of the CSV files. Notice that the last field is truncated with "..." for readability.

```
2021-07-31 22:55:43,77.212.177.88,App95,1964,EVENT,ERROR,PYV-1342,heckofa lot of issues to …
2021-07-31 22:55:43,147.101.31.147,App02,3813,REQUEST,DEBUG,UDD-5791,App92 more bugs App91 …
2021-07-31 22:55:43,237.101.33.237,App95,1198,EVENT,INFO,ZOA-3359,ERP WebLogic algorithm a …
```

The server log files present in the data lake currently span 35 days, from June 27, 2021, through July 31, 2021. For this scenario, a single log file that collects multiple servers' output is created every five minutes. Here are some more characteristics of the complete dataset.

Record size (bytes)	200
Records/ingestion file	10,000
Ingestion file size (bytes) *[see `avg_file_size` in query below]*	2,000,000
Files/day	288
Daily file size (bytes)	576,000,000
Number of days	35
Total number of ingested files *[see `tot_nbr_files` in query below]*	10,080
Total dataset size (bytes) *[see `tot_file_size` in query below]*	20,160,000,000
Total number of records	100,800,000

Step 3 - Row-oriented file formats vs. columnar file formats

A file format controls the way that each file stores the data inside it. Not all file formats store information in the same way.

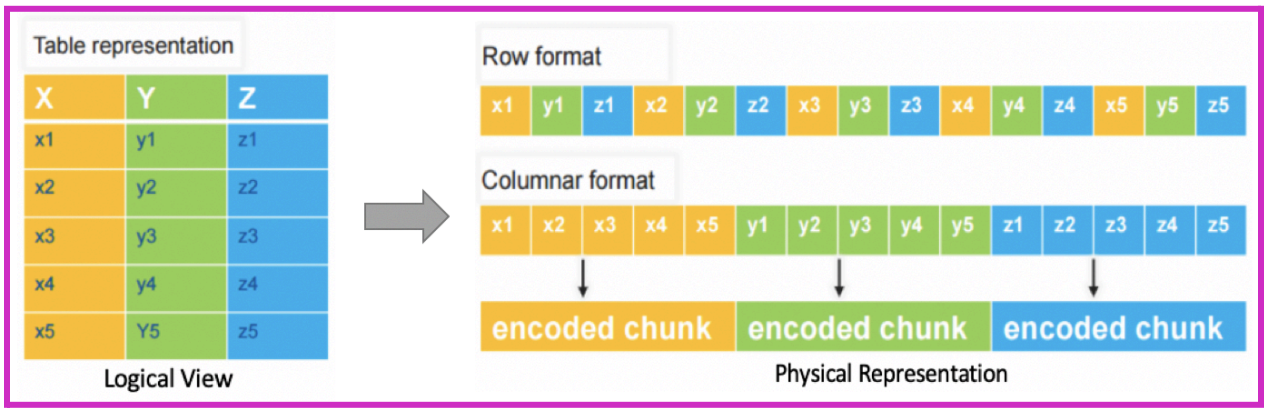
For example, the `logs_5min_ingest_csv` table is a collection of files, each holding 5 minutes worth of log data across the servers you are analyzing, and stored in a CSV file format.

The file format can make a big difference in query performance because of the underlying features of different file types. Generally speaking, data lake file formats fall into two categories.

- Row-oriented file formats
- Columnar file formats

As their names suggest, the primary difference is in how the data for a logical data table is physically stored. The row-oriented file format lines up a full record and separates it from the next one with a record delimiter.

Conversely, the columnar formats store all distinct values for a given column together, as pictured in the image below.



Columnar file formats are inherently designed for improved performance with analytics-based queries. A significant reduction in the amount of data needed to be retrieved from disk occurs when only the relevant columns are referenced.

Additional benefits include efficient encoding and compression techniques that can drastically reduce storage requirements without sacrificing query performance.

The most popular columnar file format are:

- Apache ORC
- Apache Parquet

Both of these file formats can include multiple segments of data held in a block of logical rows. Also, critical to their analytical performance benefits, both include file header and footer information. This information goes well beyond maintaining an evolution-capable schema to including persisted column-level statistics such as count, min, max and sum.

Starburst fully supports both Parquet and ORC data formats.

Step 4 - Ensure cluster is ready and metadata is cached

As this exercise is focused on verifying performance improvements, you do not want your results to be skewed by any waiting resources. The `aws-us-east-1-small` cluster being used takes time to start and any queuing time would be included in query results.

Run the following queries to ensure the environment is ready for your testing efforts. This will also cache the table's metadata.


```
SELECT *
 FROM lakehouse.serverlogs.logs_5min_ingest_csv
LIMIT 10;

SELECT *
 FROM lakehouse.serverlogs.logs_5min_ingest_orc
LIMIT 10;

SELECT *
 FROM lakehouse.serverlogs.logs_daily_rollup_orc
LIMIT 10;
```

Before you continue, verify that you are using the `aws-us-east-1-small` cluster.

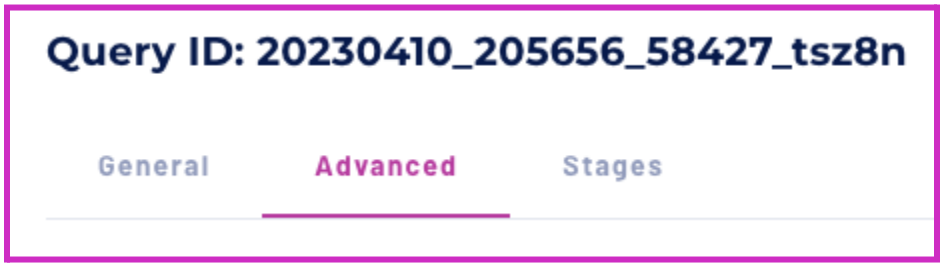
Step 5 - See reduction in bytes read with column-oriented files

Run the following query against the table that is aligned with the 10,080 CSV ingested files described above.

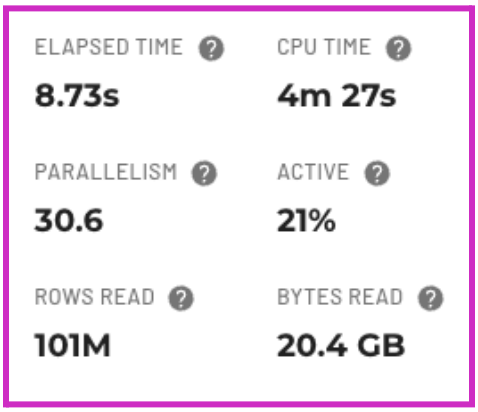
```
SELECT *
  FROM lakehouse.serverlogs.logs_5min_ingest_csv
 WHERE app_name IN ('CRM', 'ERP')
 ORDER BY app_name, log_type, log_level,
          message_id, ip_address;
```

To find some of the performance metrics you will review in this exercise, click on **View** icon in the status bar between the editor and results panes.

On the page that renders, click on the **Advanced** tab just below the **Query ID** label.



Now, locate the **Execution details** section of this page.



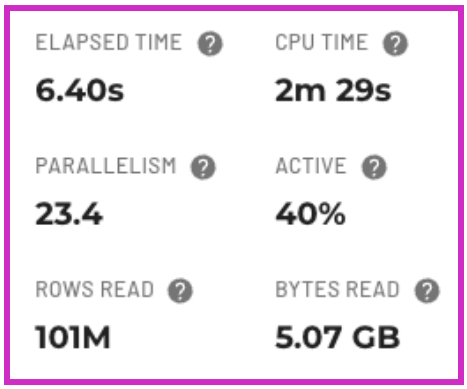
The number of **ROWS READ** and **BYTES READ** reported above align with what was presented in **Step 2**. The entire dataset was read; 20GB of data for the 101M rows of server logs.

Additionally, this cluster used ~ 4.5 minutes of **CPU TIME**. Hovering over the question mark to the right of that label explains this is the total accumulated CPU time spent on this query across all of the Workers in the cluster.

Now, run this query against another table with the same logical characteristics, but has been compacted into a fewer number of larger-sized files and persisted with the ORC file format.

```
SELECT *
  FROM lakehouse.serverlogs.logs_daily_rollup_orc
 WHERE app_name IN ('CRM', 'ERP')
 ORDER BY app_name, log_type, log_level,
          message_id, ip_address;
```


Find its **Execution details** to determine that the same number of rows were read, but the total amount of CPU time was ~ 50% using this more optimized table instead of the table aligned to small CSV files.



The same number of rows were read, but the number of bytes read are about 25% of the CSV backed table.

The primary savings in CPU time and bytes read was focused on the fact that the ORC columnar file format uses encoding and compression when being persisted and thus has much less data stored on the data lake.

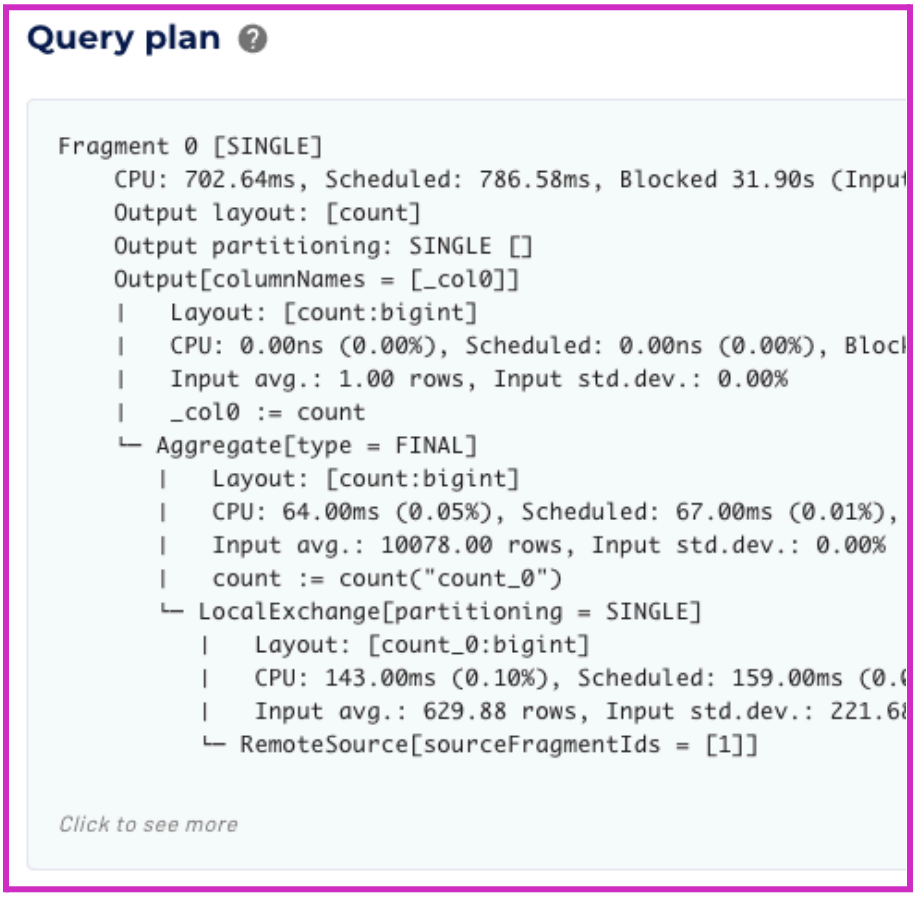
Note: You may see **ELAPSED TIME** metrics different than shown here. Variables such as the number of concurrent queries running and the cluster size can impact this. Conversely, your CPU times should be comparable to those presented. This determines the amount of resources that were needed for the unit of work to be completed.

Step 6 - Locate metrics for amount of data exchanged

This step focuses on capturing how much data is exchanged from the source stage to a later, dependent stage. This concept will make more sense when discussing query plans.

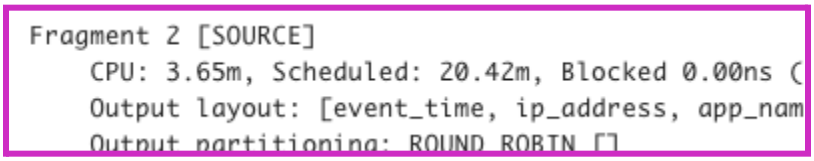
For now, you only need to understand that there is additional processing required to exchange data between stages. Similar to limiting the amount of data you read from the source system, you want to limit the data you output from the source stage.

To obtain these metrics, you will need to locate the query plan. This information is located in the same **View** icon > **Advanced** tab described in **Step 5**. Find the **Query plan** section. It should look similar to the following.



Click the **Click to see more** link at the bottom to explore the **Query Plan**.

Scroll down to the bottom to find the last stage block (identified as `Fragment n [SOURCE]`) as illustrated in the following example.



The number after `Fragment` may change from query to query, but if you remember to scroll down to the very last section, you can search the far right of the first line of the stage to locate the `Output` details.

This sample above shows that ~ 7.8M rows, with an overall size of ~ 1.5GB, is being exchanged from this source stage to whatever work is next in the query execution. You received these same values from querying both tables in **Step 5**. This is because they both are exchanging the same number of rows, with the same columns, and at this point of the query execution, the data does not matter where it was read from any longer.

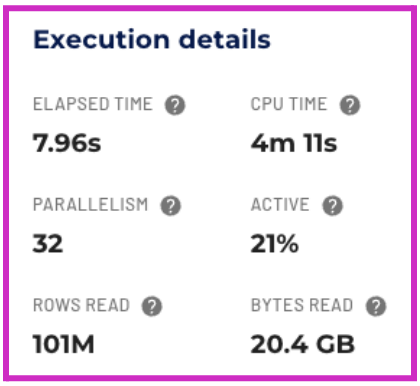
Step 7 - Projection benefits with row-oriented files

In this step, you will modify the query from **Step 5** to only retrieve a specific set of columns, instead of them all. This is called projection and always improves performance.

Run the modified query on the CSV table.

```
SELECT app_name, log_type, log_level,
       message_id, ip_address
  FROM lakehouse.serverlogs.logs_5min_ingest_csv
 WHERE app_name IN ('CRM', 'ERP')
 ORDER BY app_name, log_type, log_level,
          message_id, ip_address;
```

Retrieve this query's performance metrics.



Comparing these metrics to the same table's query execution in **Step 5** shows similar results. The row-oriented CSV file format was not able to take advantage of the limited set of columns when the data was read, but there still is a performance gain for projection with this type of file. To understand it, use the method described in **Step 6** to identify the data exchange metrics.

```
Output: 7751694 rows (443.67MB)
```

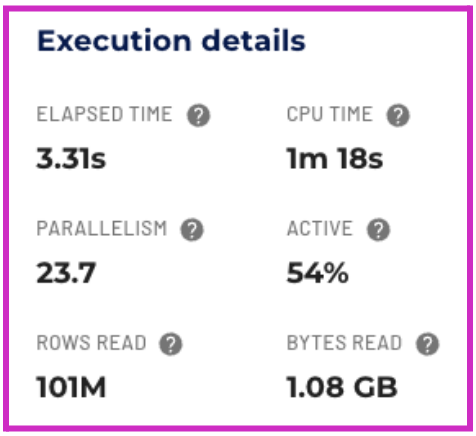
You can see that the number of rows is still ~7.8M, but the data size being exchanged is ~ 30% of the 1.5GB exchanged from the query where projection was now used. This reduction is because while the query engine could not limit the amount of data initialy read from the CSV files, it is able to eliminate the memory size of the columns that were not in the query itself by not exchanging that unnecessary data.

Step 8 - Projection benefits with column-oriented files

Modify the query from **Step 7** to use the table leveraging ORC files to determine the performance improvements available from columnar file types when using projection.

```
SELECT app_name, log_type, log_level,
       message_id, ip_address
  FROM lakehouse.serverlogs.logs_daily_rollup_orc
 WHERE app_name IN ('CRM', 'ERP')
 ORDER BY app_name, log_type, log_level,
          message_id, ip_address;
```

Capture the **Execution details**.



From using this same table back in **Step 5** we can see the same number of rows were read, but

- the bytes are now ~ 20% of the 5GB that is stored on the data lake
- The CPU time is down by about half

These significant reductions are due to column-oriented file formats allowing the query engine to only read the columns identified in the query.

Furthermore, if we compare these results with the CSV table's metrics from earlier in this step, the amount of data being read is now down to 5% (only 1GB instead of 20GB) being read from the data lake. The CPU time is down to ~ 25% as well.

The initial read from the data lake represents a large percentage of the overall resources needed for most queries to execute and this kind of change will provide significant performance improvements.

Step 9 - Locate metrics for number of splits read

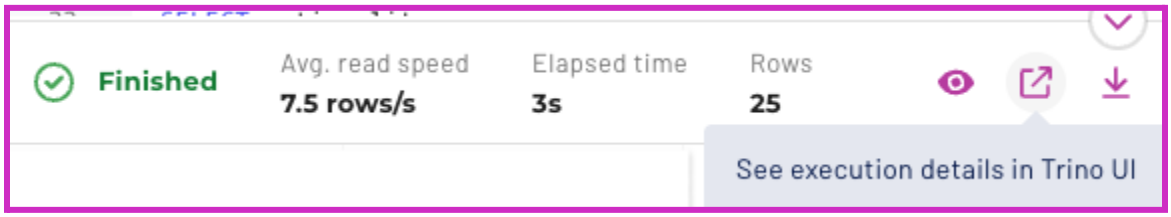
When the data lake is read, the number of files the engine has to access factors heavily into how this processing is parallelized. The comprehensive data to be read is conceptualized as a single dataset that is broken down into smaller pieces referred to as splits.

The more underlying files that are present, the more splits will be created. Moreover, each time the query engine needs to access a file there is I/O overhead. The goal is for these files to be of sufficient size (on average) so that efficiencies can be achieved by retrieving the same number of rows by accessing a smaller number of files.

Run the following query that you can use to find out how many splits were needed to read from the data lake.

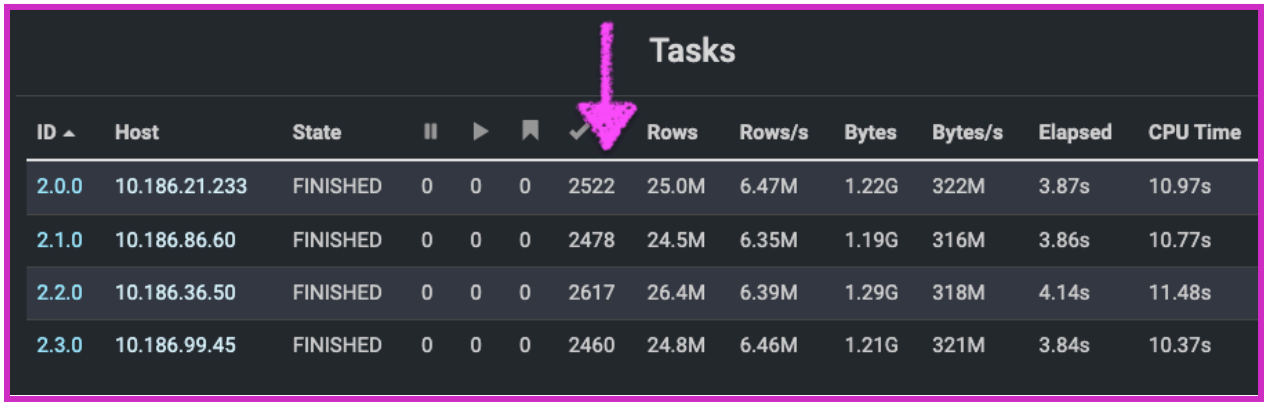
```
SELECT ip_address, log_type, log_level, message_id
  FROM lakehouse.serverlogs.logs_5min_ingest_orc
 WHERE app_name IN ('CRM', 'ERP')
   AND log_type = 'THREAT'
   AND log_level = 'FATAL'
 ORDER BY app_name, log_type, log_level, message_id;
```

In the ribbon between the query editor and the results panes, click on **Trino UI** icon to the right of the **View** icon.



This surfaces a new UI which is part of the open-source Trino project that Starburst extends from. Scroll down to the bottom of the page where the **Stages** section is located. On the very last stage number listed (i.e. the largest number) click on the down arrow icon on the far right as identified in the screenshot below.

This will provide a list of **Tasks** that participated in the query execution. For now, understand that this is the number of workers that were leveraged. In the screenshot below you can see there are four IP addresses in the **Host** column on the left. For this step, focus on the column highlighted below.



This column with the checkmark heading represents the number of splits that were processed by each of the workers. Added those up you can see there were approximately 10K splits.

Step 10 - Validate small files problems

Prior steps have compared `logs_5min_ingest_CSV` (small, row-oriented, files) with `logs_daily_rollup_orc` (compacted files using columnar files) to show performance

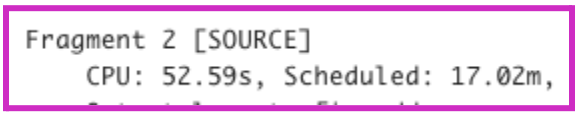
gains. In this step, you will use another table named `logs_5min_ingest_ORC` in place of `logs_5min_ingest_CSV` to validate the small files problem more clearly by having the only variable being file size.

Importantly, this table is a mixture of the other two. On the one hand, It is uncompacted, having the same number of files as the CSV table, and the files themselves are still small; on the other hand, it uses the ORC file format. Because the number of small files is the same as the CSV table, this is a good test case to show the benefit of eliminating them.

The query used in **Step 9** against this ORC small files table showed that there was a split for each of the individual files.

Now, let's take a moment to observe the query plan from that query execution. If you need a reminder how to locate the query plan, refer back to the instructions in **Step 6**.

Recall that query plans are read from the bottom up. In Fragment 2 notice the first two metrics presented.

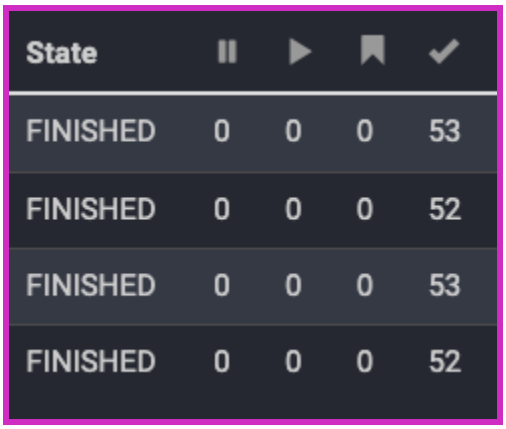


The **CPU** metric listed is the same CPU time referenced previously and records actual CPU time spent. In contrast, the **Scheduled** metric identifies how much CPU time was reserved for data lake read operations in this case. This value shows 17x more time than was actually devoted to processing data than was used, suggesting that the actual processing time was much smaller than the time reserved. During this time, CPUs are in standby, waiting for I/O operations. You can see that substantial savings would be possible if we could narrow the gap between scheduled time and actual time.

Now it's time to run **Step 9** using the new ORC file. Use the code below to switch to the compacted version of the data.

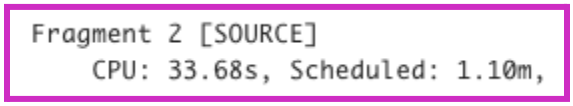
```
SELECT ip_address, log_type, log_level, message_id
  FROM lakehouse.serverlogs.logs_daily_rollup_orc
 WHERE app_name IN ('CRM', 'ERP')
   AND log_type = 'THREAT'
   AND log_level = 'FATAL'
 ORDER BY app_name, log_type, log_level, message_id;
```


Now follow the rest of **Step 9** and observe the number of splits being processed on this table. Your output should be similar to the image below which shows the small files table has two orders of magnitude more files (10,000 vs 100).



State				
FINISHED	0	0	0	53
FINISHED	0	0	0	52
FINISHED	0	0	0	53
FINISHED	0	0	0	52

Determine the CPU and Scheduled metrics for this query.



```
Fragment 2 [SOURCE]
    CPU: 33.68s, Scheduled: 1.10m,
```

You can see that the actual amount of time to process the data was about 50% as much, but noticed the amount of time the CPUs were devoted to reading data is 1/17th as small. This 17x increase means that all that extra CPU waiting time cannot be allocated to other activities.

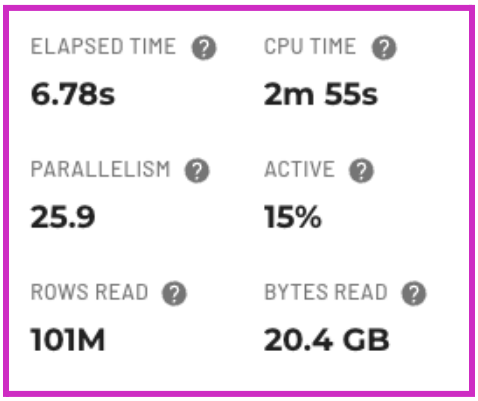
Step 11 - Embedded column statistics benefits for filtering

The column-oriented file formats like ORC and Parquet store column-level statics for the records in each individual file. This can be leveraged via a type of predicate pushdown by allowing the query engine to first read the small number of bytes from a file that represent the statistics.

At that point, the query engine can determine if the file in question might, or definitely won't, have the data it is looking for. For simple filters this can be explained by seeing if the min & max values for a particular column are aligned with the `WHERE` clause. To see this in practice, enhance the prior query to have an additional set of filters on the `event_time` column and switch back to the CSV table which will not have the benefit of these column statistics.

```
SELECT ip_address, log_type, log_level, message_id
  FROM lakehouse.serverlogs.logs_5min_ingest_csv
 WHERE app_name IN ('CRM', 'ERP')
   AND log_type = 'THREAT'
   AND log_level = 'FATAL'
   AND event_time >=
         from_iso8601_timestamp(
               '2021-07-02T00:00:00')
   AND event_time <
         from_iso8601_timestamp(
               '2021-07-05T00:00:00')
 ORDER BY app_name, log_type, log_level, message_id;
```

Determine the **Execution details** metrics.

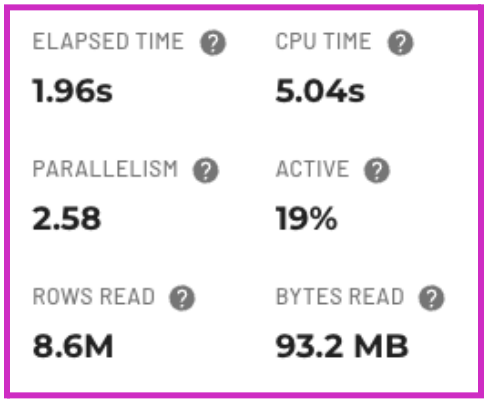


All rows were read from the 20GB combined file size for this table. It consumed ~ 3 minutes of CPU time.

Run the query again against the ORC table.

```
SELECT ip_address, log_type, log_level, message_id
  FROM lakehouse.serverlogs.logs_daily_rollup_orc
 WHERE app_name IN ('CRM', 'ERP')
   AND log_type = 'THREAT'
   AND log_level = 'FATAL'
   AND event_time >=
         from_iso8601_timestamp(
               '2021-07-02T00:00:00')
   AND event_time <
         from_iso8601_timestamp(
              '2021-07-05T00:00:00')
ORDER BY app_name, log_type, log_level, message_id;
```

Determine the **Execution details** metrics.



Since the three data files for each day were logically sorted by the event_time field, simply due to how they were ingested & compacted, the query engine was able to determine only 15 out of 105 files needed to be fully read. All the other files were abandoned when it was determined their min & max values were not aligned with the July 2-3 filter in the query.

This resulted in < 9M rows being read and < 100MB of actual data. This dramatic reduction in data needed to be read resulted in only requiring 3% of the CPU time than the row-oriented file format needed.

Step 12 - Embedded column statistics benefits for aggregations

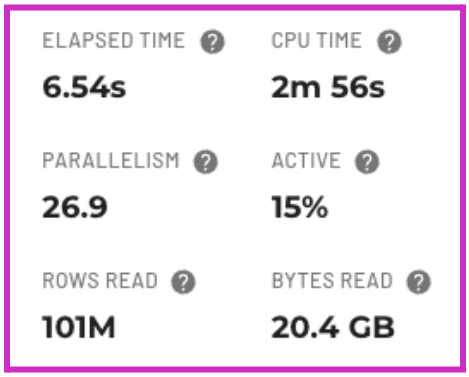
The column-level statistics stored in each column-oriented file also benefits aggregation queries. Simple MIN or MAX calculations for a GROUP BY clause simply leverage the embedded statistics described in the prior step by collecting them from all files and quickly determining the correct value.

Even calculations such as AVG and SUM are benefited by additional embedded values for each file. The query engine can get details from each file and much more quickly generate the final aggregation-based calculation.

Run this query on the CSV table.

```
SELECT app_name, count() as num_events
  FROM lakehouse.serverlogs.logs_5min_ingest_CSV
 GROUP BY app_name
 ORDER BY num_events, app_name;
```

Determine the **Execution details** metrics.

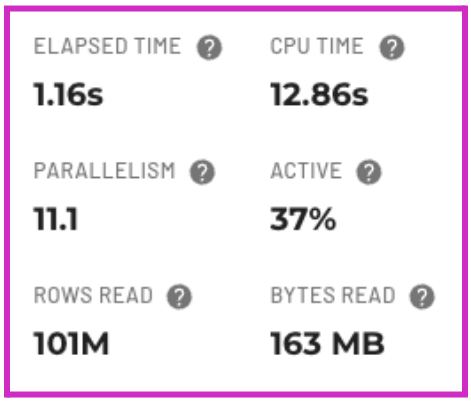


These results are very similar to those on the same table in **Step 11** as the entire dataset had to be read.

Run the query again against the ORC table.

```
SELECT app_name, count() as num_events
  FROM lakehouse.serverlogs.logs_daily_rollup_orc
 GROUP BY app_name
 ORDER BY num_events, app_name;
```

Determine the **Execution details** metrics.



Note that it only required ~ %7 of the CPU time on the ORC table.

Summary

- You reviewed the differences between row-oriented and columnar file formats which start with much smaller number of bytes needed due to encoding and compression
- You determined how small files create performance issues because of significant CPU waiting time on I/O operations
- Regardless of file format types you validated the benefit of the project & filter – early & often mantra
- You realized the further performance benefits of projection and filtering with column-oriented file format
- You understand how column statistics embedded within individual file improves performance with filtering and aggregations

END OF LAB EXERCISE

Lab 3: Exploring table partitioning and bucketing

Estimated completion time

- 40 minutes

Learning objectives

- In this lab, you will explore how table partitioning and bucketing, used independently or together, can improve performance. You will see how both of these strategies will read far less data than a full scan of a table when there is a filter that aligns appropriately. You will understand the common strategy for creating a partition column that is derived from a more finite timestamp value.

Prerequisites

- Student setup - Lab 1: Create student account

Activities

1. Prepare for the exercise
2. Review partitioning & bucketing functionality
3. Create a data lake table from TPCH orders table
4. Locate metrics for data read
5. Table scan orders searching by priority
6. Create an orders table partitioned on priority
7. Identify partition folders that were read
8. Leverage partition pruning with partitioned orders table
9. Using a partition column derived from a timestamp
10. Table scan orders searching by customer
11. Create an orders table bucketed on customer
12. Read fewer files with bucketed orders table

Step 1 - Prepare for the exercise

- Sign in and verify the `students` role is selected in the upper-right corner.
- Ensure the `aws-us-east-1-free` **Cluster** is reporting a **Status** of `Running`.
- In the **Query editor**, select `aws-us-east-1-free` in the cluster pulldown.

Note: *If you did not previously create a schema*, execute the following SQL statement. Make sure that you replace `yourname` with your actual name, or another identifier that you prefer. **Only use lowercase characters and numbers; no special characters or spaces.**

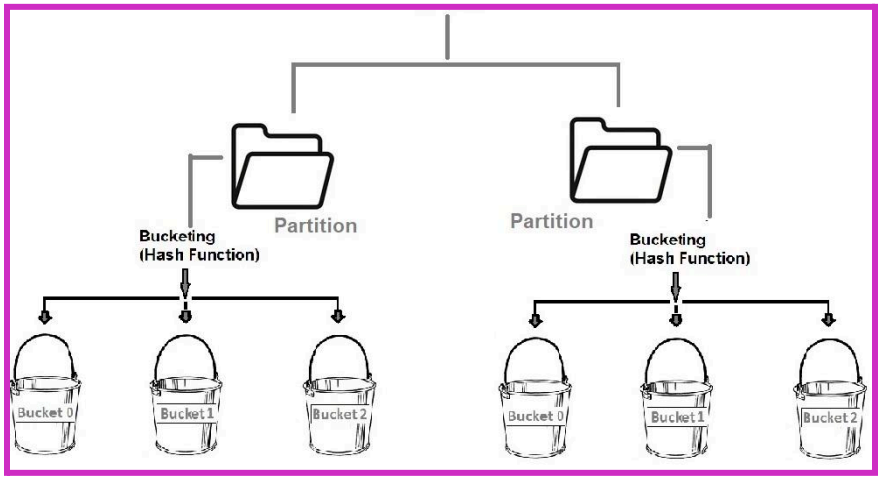
```
CREATE SCHEMA students.yourname;
```

Step 2 - Review partitioning & bucketing functionality

Partitioning is just a way to divide data into subfolders. The field(s) you partition a table on become subfolders of the table's data lake folder. When a `WHERE` clause is aligned to the

partitioned field, then only the folder(s) that meet the particular criteria are read – the rest are ignored.

With bucketing you choose the number of buckets up front and this number if fixed. The values for each row for the bucketed field determine which bucket (a file) the row will consistently go into. When a `WHERE` clause is aligned to the bucketed field, then only the the appropriate file(s) are read – the rest are ignored.



Partitioning and bucketing can be used together or by themselves.

Note: Be careful when creating small files using these features.

Step 3 - Create a data lake table from TPCH orders table

Run a CTAS statement to make a data lake table from the data present in the TCPH orders table.

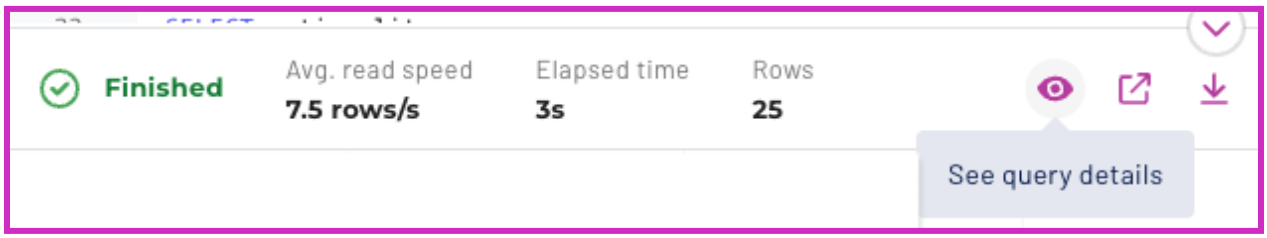
```
USE students.yourname;


CREATE TABLE orders_no_pttn
   WITH (
      type = 'hive'
   )
AS
   SELECT *
     FROM tpch.sf1.orders;
```


Verify that the following query returns 941 rows.

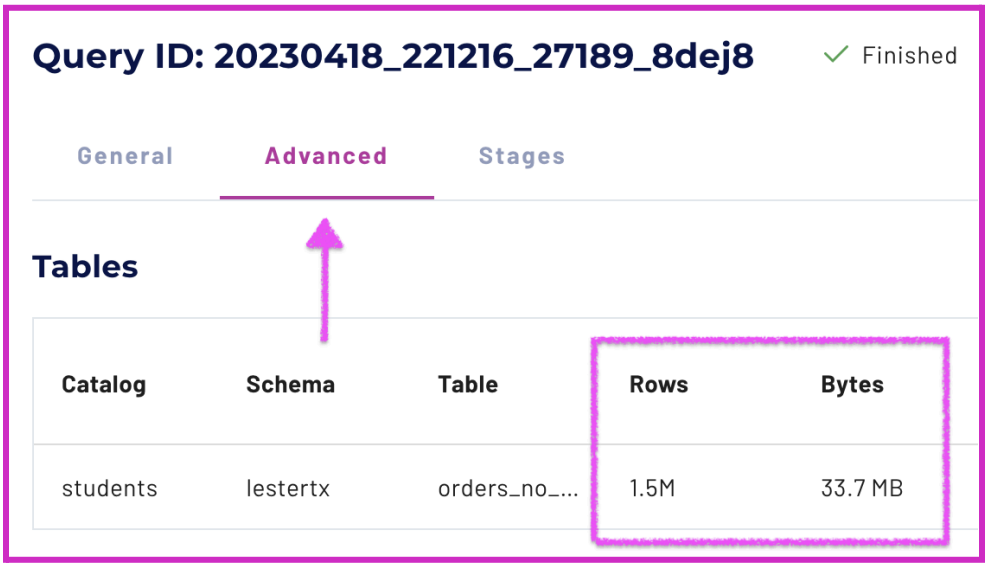
```
SELECT * from orders_no_pttn
 WHERE totalprice >= 430000.00;
```

Step 4 - Locate metrics for data read

Click the **Query details** icon between the query editor pane and the query results pane.



Click on the **Advanced** tab below the **Query ID:** label.



In the **Tables** section of the **Advanced** tab you will see values for number of **Rows**, and the **Bytes**, read.

For the last query in **Step 3**, all rows (1.5M) were read from the `order_no_pttn` table which accounted for 33.7MB of data. For this table, that equated to a full table scan of all records from the data lake.

Step 5 - Table scan orders searching by priority

Enhance the query filtering on the order's priority field with the following query that returns 187 rows.

```
SELECT * from orders_no_pttn
 WHERE totalprice >= 430000.00
   AND orderpriority = '4-NOT SPECIFIED';
```

Using the process described in **Step 4**, verify that a full table scan occurred of all 1.5M rows and 33.7MB of data.

Step 6 - Create an orders table partitioned on priority

To create another version of the orders table using the priority field for the partitioning strategy, modify the CTAS statement from **Step 3**. Enhance it with the appropriate `partitioned_by` value.

```
CREATE TABLE orders_pttn_priority
   WITH (
      partitioned_by = ARRAY['orderpriority'],
      type = 'hive'
   )
AS
   SELECT *
     FROM tpch.sf1.orders;
```

You will receive the following error.

> Partition keys must be the last columns in the table and in the same order as the table properties: [orderpriority]

Just as it states, the partition field needs to be the last column in this (Hive) table.

Execute the following statement that reorders the columns so that `orderpriority` is last.

```
CREATE TABLE orders_pttn_priority
   WITH (
      partitioned_by = ARRAY['orderpriority'],
      type = 'hive'
   )
AS
   SELECT orderkey, custkey, orderstatus, totalprice,
          orderdate, clerk, shippriority, comment,
          orderpriority
     FROM tpch.sf1.orders;
```

Step 7 - Identify partition folders that were read

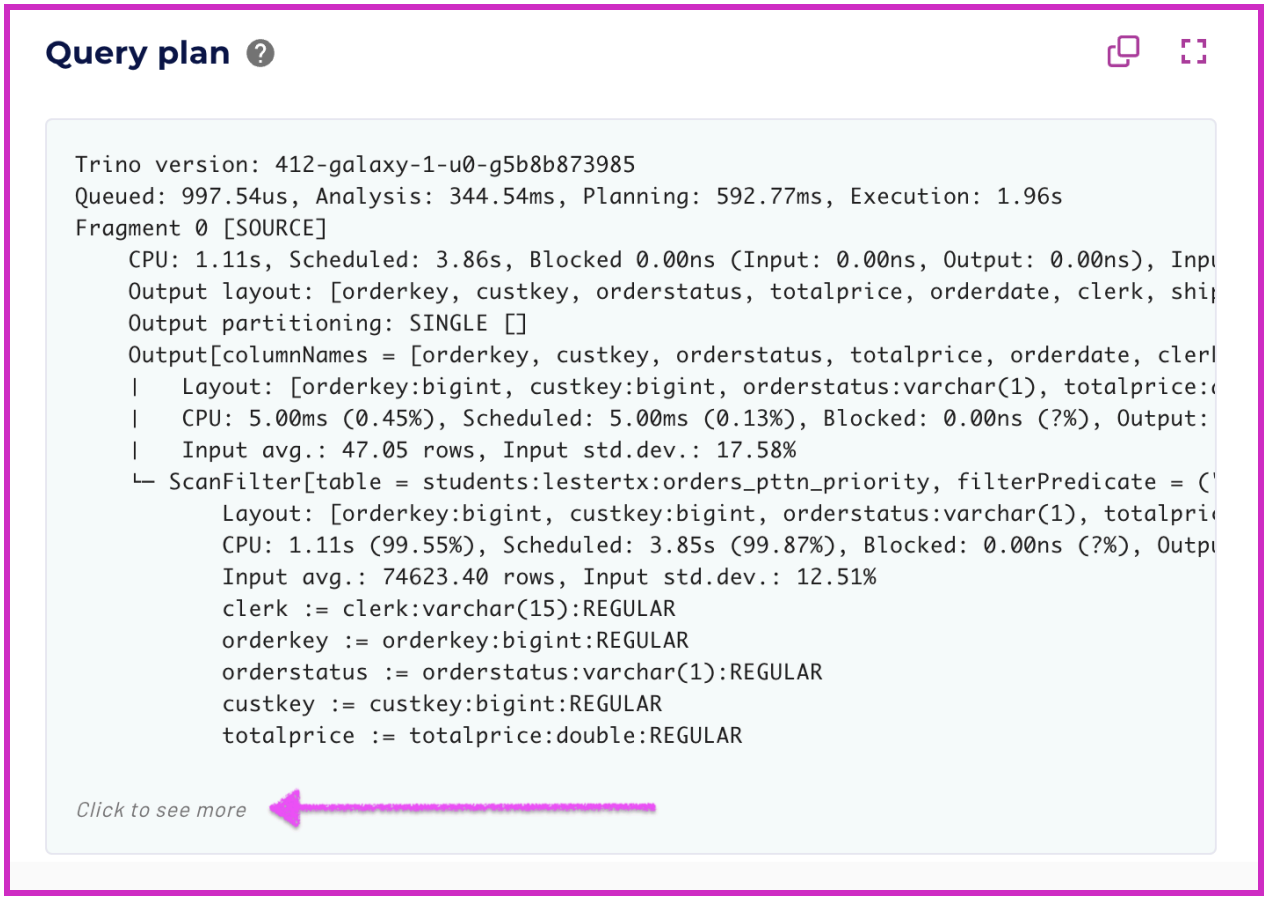
Run the original query from Step 3, but on this new partitioned table.

```
SELECT * from orders_pttn_priority
 WHERE totalprice >= 430000.00;
```

Using the process described in **Step 4**, verify that a full table scan occurred of all 1.5M rows and with a bytes read value similar to the non-partitioned table.

Rows	Bytes
1.5M	34.8 MB

This means that all partition folders for the distinct values of the partitioned field were all read. To verify, look for the **Query plan** section on the same page as you are now on. If you navigated away, click on **Query details** then the **Advanced** tab.



Now, click on the **Click to see more** link at the bottom of the **Query plan** section as highlighted above.

Focus on the two lines above the final output line. These should start with the `partitioned_by` column name.

Output

```
orderpriority := orderpriority:varchar(15):PARTITION_KEY
     :: [[1-URGENT], [2-HIGH], [3-MEDIUM], [4-NOT SPECIFIED],
[5-LOW]]
```

Because this is a partitioned column, the query plan will show you the folders that were read. The five above represent all of the distinct values for the `orderpriority` column. This is further evidence that a full table scan occurred.

Step 8 - Leverage partition pruning with partitioned orders table

Add an `orderpriority` filter to the query and execute it.

```
SELECT * from orders_pttn_priority
 WHERE totalprice >= 430000.00
   AND orderpriority = '4-NOT SPECIFIED';
```

Determine how many rows and bytes were read while executing this query.

Rows	Bytes
297K	6.96 MB

This represents about 20% of the full table scan's metrics. The sample data we used from TPCH was relatively evenly distributed across the five order priority values which provides anecdotal evidence that only one of the partition folders was read.

Using the process described in **Step 7**, determine which folder(s) were read during this query's execution.

Output

```
orderpriority := orderpriority:varchar(15):PARTITION_KEY
    :: [[4-NOT SPECIFIED]]
```

This solidifies the anecdotal evidence with actual results. When the query engine can read a subset of the partition folders, this is referred to as *partition pruning*.

Step 9 - Using a partition column derived from a timestamp

To make the benefits of partitioning more obvious, this step will utilize a larger table holding 35 days of server logs with 101M rows.

Run the following to understand how this partitioned table was created.

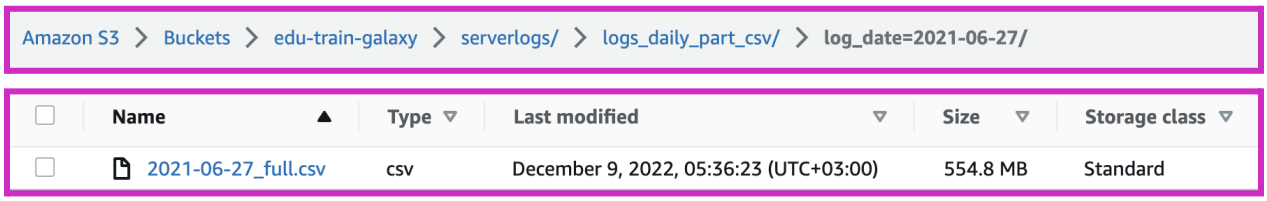
```
SHOW CREATE TABLE lakehouse.serverlogs.logs_daily_part_csv;
```

Output

```
CREATE TABLE lakehouse.serverlogs.logs_daily_part_csv (
   event_time timestamp(3),
   ip_address varchar(15),
   app_name varchar(25),
   process_id smallint,
   log_type varchar(15),
   log_level varchar(15),
   message_id varchar(15),
   message_details varchar(555),
   log_date char(10)
)
WITH (
   external_location =
        's3://edu-train-galaxy/serverlogs/logs_daily_part_csv',
   format = 'TEXTFILE',
   partitioned_by = ARRAY['log_date'],
   textfile_field_separator = ',',
   type = 'HIVE'
)
```

Note: You can see the table is partitioned on the `log_date` field, but there is also an event_time field. It is a common practice with Hive tables that need to be partitioned on a more coarse-grained representation of a timestamp to create a second field like this `log_date` example. That is to say that the `log_date` will always be the same for any record with an `event_time` for that same day.

You do not have access to review the data in S3, but here is an example of the contents of one of the 35 days (June 27 - July 31, 2021) as stored in a partition folder.



Each of the 35 days has a single file of ~ 555MB in size in their respective partition folders. This totals up to ~ 20GB of data.

Run a query attempting to leverage only three of the 35 days of data.

```
SELECT ip_address, log_type, log_level
  FROM lakehouse.serverlogs.logs_daily_part_csv
 WHERE app_name IN ('CRM', 'ERP')
   AND log_type = 'THREAT'
   AND log_level = 'FATAL'
   AND event_time >=
         from_iso8601_timestamp(
               '2021-07-02T00:00:00')
   AND event_time <
         from_iso8601_timestamp(
               '2021-07-05T00:00:00')
 ORDER BY app_name, ip_address;
```

Determine how many records and bytes were read.

Rows	Bytes
101M	20.4 GB

This indicates that a full table scan was performed.

Determine which directories were read.

Output

```
log_date:char(10):PARTITION_KEY
    :: [[2021-06-27, 2021-07-31]]
```

This shows that an array of directories from 2021-06-27 through 2021-07-31 were read. This verifies that the entire data set was searched through.

The full table scan occurred as the query used the `event_time` column, not the partitioned `log_date` column.

Revise the query to use the column that will be leveraged more efficiently.

```
SELECT ip_address, log_type, log_level
  FROM lakehouse.serverlogs.logs_daily_part_csv
 WHERE app_name IN ('CRM', 'ERP')
   AND log_type = 'THREAT'
   AND log_level = 'FATAL'
   AND log_date >= '2021-07-02'
   AND log_date <= '2021-07-04'
 ORDER BY app_name, ip_address;
```

Determine how many records and bytes were read.

Rows	Bytes
8.6M	1.75 GB

The 101M rows and 20GB of data are relatively evenly distributed across the 35 days. The query was expected to read three of the 35 folders (3/35 or 8.5% of the full data). Performing some quick calculations provides anecdotal evidence that only three of the partition folders were read.

Determine which folders were read during this query's execution.

Output

```
log_date:char(10):PARTITION_KEY
    :: [[2021-07-02], [2021-07-03], [2021-07-04]]
```

This solidifies the anecdotal evidence with actual results. The partition pruning resulted in only the three specific day's folders being read.

Step 10 - Table scan orders searching by customer

Now, run the following query on the non-partitioned orders table to see that searching by a particular customer will result in a full table scan.

```
SELECT *
  FROM orders_no_pttn
 WHERE custkey = 130000;
```

Verify that all 1.5M rows were read.

Rows	Bytes
1.5M	33.7 MB

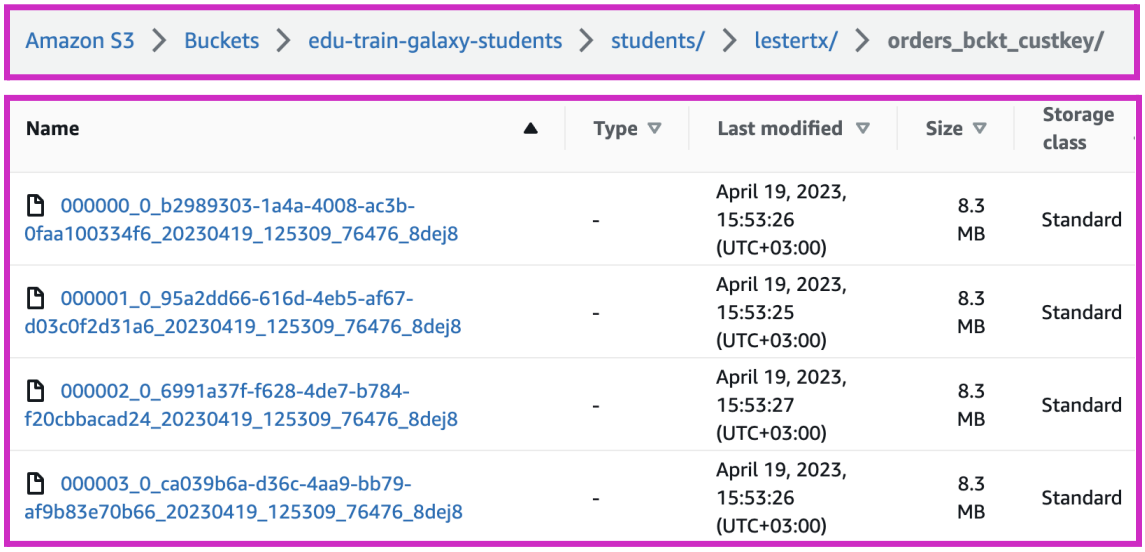
Step 11 - Create an orders table bucketed on customer

Create another table with the same data, but bucket it on the customer identifier column.

```
CREATE TABLE orders_bckt_custkey
   WITH (
      bucketed_by = ARRAY['custkey'],
      bucket_count = 4,
      type = 'hive'
   )
AS
   SELECT *
     FROM tpch.sf1.orders;
```

You do not have access to review the data in S3, but here is an example of the contents of this table. The data was even spread by custkey into four separate files of similar size.

Adding them all up validates that a full table scan would read a similar byte count as seen in **Step 10**.

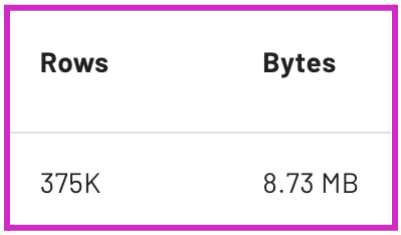


Step 12 - Read fewer files with bucketed orders table

Run the query again, but against the bucketed table.

```
SELECT *
  FROM orders_bckt_custkey
 WHERE custkey = 130000;
```

Determine how much data was read.



Only about 25% of the data was read because the specific `custkey` value could only be in one of the four buckets on the data lake.

The customer identifier would not have been a good partitioning column because there are so many unique customers and each customer only has a few orders. That would create a very small file problem. Bucketing was a better choice for this kind of performance expectation.

Remember, partitioning and bucketing can be used independently or in combination.

Summary

- You reviewed the functionality of partitioning & bucketing and how they can be used independently or together
- You determined how to tell if a full table scan was executed to complete a query
- You created a partitioned table that was designed for a particular filter
- You determined how to see if partition pruning worked as expected
- You explored strategies for when a partition needs to be created based on a more coarse-grained aspect of a timestamp
- You created a bucketed table that was designed for a particular filter
- You realized that the bucketed table only reads a single bucket/file when filtering on the bucketed column

END OF LAB EXERCISE

Table formats

Lab 1: Explore the Delta Lake table format

Estimated completion time

- 30 minutes

Learning objectives

- In this lab you will explore the concept of table formats by learning about Delta Lake. You will create delta lake tables, modify them, and observe changes in the corresponding Delta Log, which records all additions, deletions, and updates made to a Delta Lake table. You will end by investigating how Delta Log files are stored and what kind of information you can gather by looking inside them.

Prerequisites

- Student setup - Lab 1: Create student account

Activities

1. Prepare for the exercise
2. Create Delta Lake tables
3. Verify creation of Delta Log
4. Modify table
5. Observe Delta Log changes
6. Inspect Delta Log file
7. Add three additional records
8. Update rows with additional ID data
9. Understanding how the Delta Log enacts updates
10. Understanding Deltas
11. Understanding how Delta Log tracks deletions

Step 1 - Prepare for the exercise

- Sign in and verify the `students` role is selected in the upper-right corner.
- Ensure the `aws-us-east-1-free` **Cluster** is reporting a **Status** of `Running`.
- In the **Query editor**, select `aws-us-east-1-free` in the cluster pulldown.

Note: *If you did not previously create a schema*, execute the following SQL statement. Make sure that you replace `yourname` with your actual name, or another identifier that you prefer. **Only use lowercase characters and numbers; no special characters or spaces.**

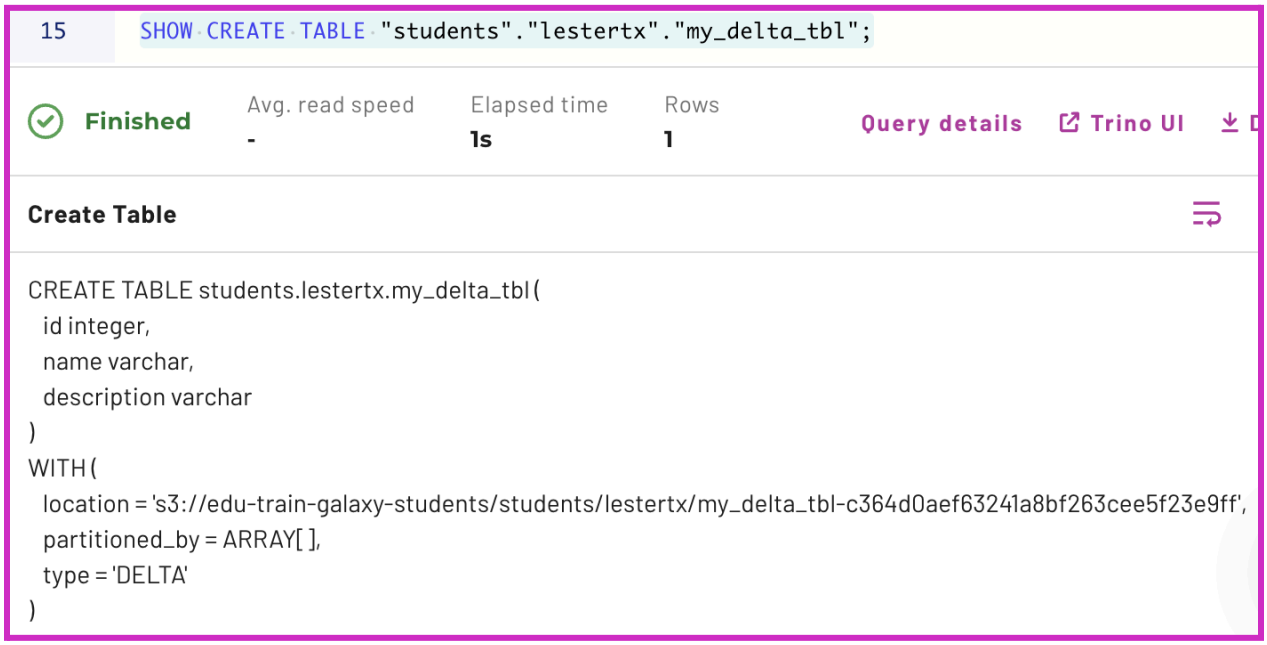
```
CREATE SCHEMA students.yourname;
```

Step 2 - Create Delta Lake tables

Create a new table in your schema named `my_delta_tbl` as shown below. Notice the `WITH` clause ensuring that it is created with the Delta Lake type.

```
USE students.yourname;


CREATE TABLE my_delta_tbl (
   id integer,
   name varchar(55),
   description varchar(255)
) WITH (
   type = 'delta'
);
```

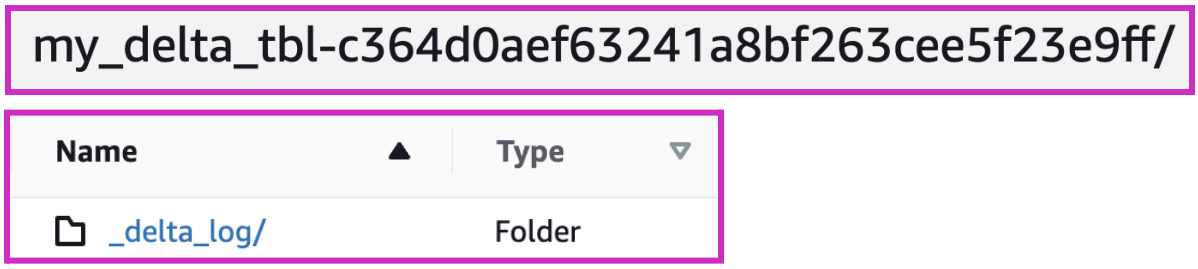
Your `SHOW CREATE TABLE` output should look similar to this.



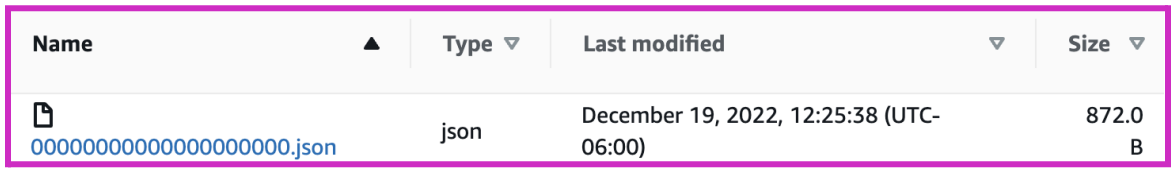
Step 3 - Verify creation of Delta Log

Now it's time to verify that you have successfully created the table.

You cannot see the S3 object store, but there is a folder created for your table that only has a `_delta_log` subdirectory at this time.



Inside the `_delta_log` folder there is a single JSON file.



Note: This exercise will describe some of the key elements in the metadata files that are created. The primary goals are to give you an awareness of the complexity of Delta Lake as well as present you with a starting point for additional research beyond this exercise. You are not expected to fully understand the exact mechanisms used by Delta Lake from just this exercise.

The JSON file's contents look similar to this.


```
{"commitInfo":{"version":0,"timestamp":1671474337657,"userId":"lester
_tx@hotmail.com","userName":"lester_tx@hotmail.com","operation":"CREA
TE
TABLE","operationParameters":{"queryId":"20221219_182536_61616_zhfuq"
},"clusterId":"trino-403-58-g8595fb7-trino-demo-coordinator-6877fb47d
8-74qf4","readVersion":0,"isolationLevel":"WriteSerializable","blindA
ppend":true}}

{"protocol":{"minReaderVersion":1,"minWriterVersion":2}}

{"metaData":{"id":"93ddb64c-bfae-415d-bcc3-dd9c05f2d236","format":{"p
rovider":"parquet","options":{}},"schemaString":"{\"fields\":[{\"meta
data\":{},\"name\":\"id\",\"nullable\":true,\"type\":\"integer\"},{\"
metadata\":{},\"name\":\"name\",\"nullable\":true,\"type\":\"string\"
},{\"metadata\":{},\"name\":\"description\",\"nullable\":true,\"type\
":\"string\"}],\"type\":\"struct\"}","partitionColumns":[],"configura
tion":{},"createdTime":1671474337657}}
```


This log file indicates that the table was newly created. It also shows the schema for the table.

Step 4 - Modify table

The Delta Log tracks changes in the table. You can think of it as a log entry containing the essential details of the table, including its creation and schema.

Add a few records using the code below.

```
INSERT INTO
 my_delta_tbl (id, name, description)
VALUES
 (1, 'one', 'added via 1st INSERT statement'),
 (2, 'two', 'added via 1st INSERT statement'),
 (3, 'three', 'added via 1st INSERT statement');
```

Step 5 - Observe Delta Log changes

The changes made to the table have updated the Delta Log. These changes are saved in a JSON file in your Delta Log directory.

In S3, the table's base folder now has an actual data file alongside the `_delta_log` subdirectories. The file will look similar to the one pictured below. This file contains the three rows just inserted.

Name	Type	Last modified	Size
_delta_log/	Folder	-	-
20221219_203215_26025_zhfuq-194960dd-b0df-4306-af2b-5d86c704164a	-	December 19, 2022, 14:32:19 (UTC-06:00)	613.0 B

Inside `_delta_log`, there is now a second JSON file.

Name	Type	Last modified	Size
00000000000000000000.json	json	December 19, 2022, 12:25:38 (UTC-06:00)	872.0 B
00000000000000000001.json	json	December 19, 2022, 14:32:20 (UTC-06:00)	801.0 B

Step 6 - Inspect Delta Log file

The additional log file contains a record of the changes to the table, saved in JSON.


```
{"commitInfo":{"version":1,"timestamp":1671481938851,"userId":"lester
_tx@hotmail.com","userName":"lester_tx@hotmail.com","operation":"WRIT
E","operationParameters":{"queryId":"20221219_203215_26025_zhfuq"},"c
lusterId":"trino-403-58-g8595fb7-trino-demo-coordinator-7c8b66fdc5-hz
pf9","readVersion":0,"isolationLevel":"WriteSerializable","blindAppen
d":true}}

{"add":{"path":"20221219_203215_26025_zhfuq-194960dd-b0df-4306-af2b-5
d86c704164a","partitionValues":{},"size":613,"modificationTime":16714
81938422,"dataChange":true,"stats":"{\"numRecords\":3,\"minValues\":{
\"id\":1,\"name\":\"one\",\"description\":\"added via 1st INSERT
statement\"},\"maxValues\":{\"id\":3,\"name\":\"two\",\"description\"
:\"added via 1st INSERT
statement\"},\"nullCount\":{\"id\":0,\"name\":0,\"description\":0}}",
"tags":{}}}
```


The information inside the Delta Log file can be useful. For instance, this log file indicates that new data was written to the table and calls out the particular file name where the three new records are stored.

Step 7 - Add three additional records

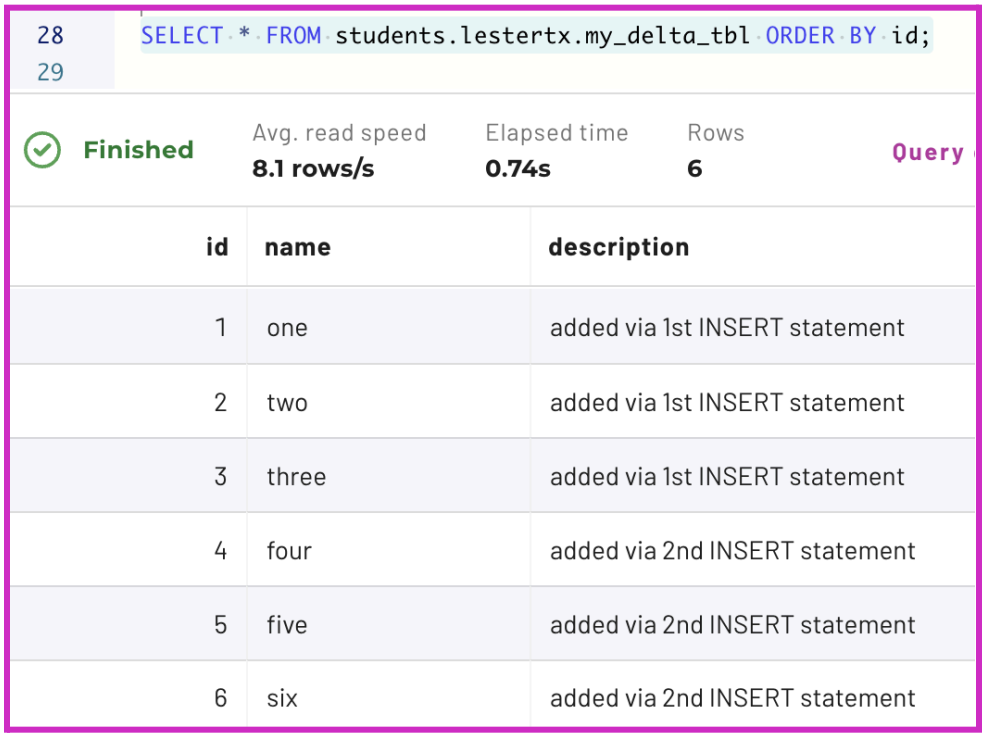
Test your Delta Log by adding a few more records using the code below.

Note: These are all different values from the prior `INSERT` statement.

```
INSERT INTO
 my_delta_tbl (id, name, description)
VALUES
 (4, 'four', 'added via 2nd INSERT statement'),
 (5, 'five', 'added via 2nd INSERT statement'),
 (6, 'six', 'added via 2nd INSERT statement');
```

Verify that all six rows are present.

```
SELECT * FROM my_delta_tbl ORDER BY id;
```



id	name	description
1	one	added via 1st INSERT statement
2	two	added via 1st INSERT statement
3	three	added via 1st INSERT statement
4	four	added via 2nd INSERT statement
5	five	added via 2nd INSERT statement
6	six	added via 2nd INSERT statement

The addition of the new records causes a new Delta Log file to be created in S3. Remember that this Delta Log file is saved in addition to a new data file within the new records themselves. The Delta Log is metadata used to track changes in the table itself.

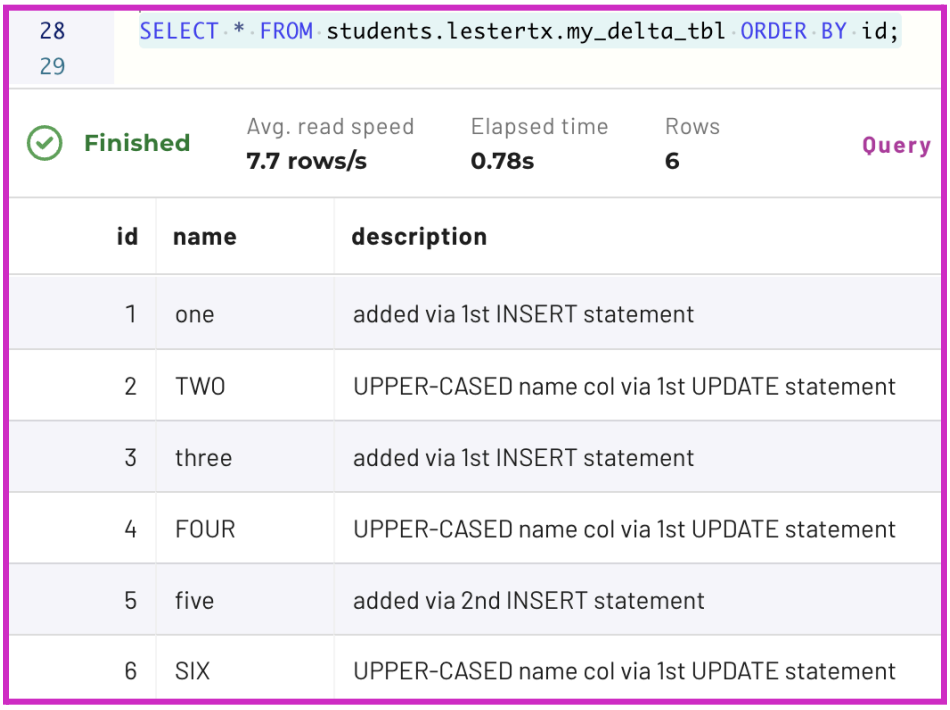
Step 8 - Update rows with additional ID data

Delta Logs track updates as well as additions. To test this, update the rows with an even `id` value which should update 3 of the 6 rows.

```
UPDATE
 my_delta_tbl
SET
 description = 'UPPER-CASED name col via 1st UPDATE statement',
 name = UPPER(name)
WHERE
 (id % 2) = 0;
```

Your results should be similar to those pictured below.

```
SELECT * FROM my_delta_tbl ORDER BY id;
```



id	name	description
1	one	added via 1st INSERT statement
2	TWO	UPPER-CASED name col via 1st UPDATE statement
3	three	added via 1st INSERT statement
4	FOUR	UPPER-CASED name col via 1st UPDATE statement
5	five	added via 2nd INSERT statement
6	SIX	UPPER-CASED name col via 1st UPDATE statement

Step 9 - Understanding how the Delta Log enacts updates

The information held inside Delta Log files is used to track all updates made to the table. Although some of the information inside these files is beyond the scope of this module, the ways that this metadata is handled are important to bear in mind.

Most notably, Delta Log updates records in a very particular way. Updating an individual row does not update the row in question on a record-for-record basis. Instead, it records a series of events in the Delta Log which collectively result in an update. These include:

- One, or more, record identifying the removal of any files that were affected by the update
- One, or more, record identifying the addition of any new files containing the updated data

In this way, Delta Log achieves update functionality without having to enact a traditional in-place update.

How does this work in practice? Consider the example below, which shows the process that Delta Lake uses to perform an update.

A - Identify update


```
{"commitInfo":{"version":3,"timestamp":1682028661424,"userId":"lester
_tx@hotmail.com","userName":"lester_tx@hotmail.com","operation":"MERG
E","operationParameters":{"queryId":"20230420_221059_14321_vy3rm"},"c
lusterId":"trino-412-galaxy-1-u0-g5b8b873985-trino-demo-coordinator-6
d7fd7b674-qcvlc","readVersion":2,"isolationLevel":"WriteSerializable"
,"isBlindAppend":true}}
```


In line **A** of this log file, there is a declaration of a `MERGE` being made. This tells Delta Lake that it should begin the update process.

B - Identify file for removal


```
{"remove":{"path":"20221219_203919_31070_zhfuq-ef4f0055-58b4-45e0-95c
2-0ccac4d6928c","deletionTimestamp":1671482883029,"dataChange":true}}
```


C - Identify file for removal


```
{"remove":{"path":"20221219_203215_26025_zhfuq-194960dd-b0df-4306-af2
b-5d86c704164a","deletionTimestamp":1671482883029,"dataChange":true}}
```


Lines **B** and **C** identify the change that will take place. Specifically, the two files created through `INSERT` commands earlier are flagged for removal. You can see information about the files, as well as the timestamp associated with them.

D - Identify replacement file to be added


```
{"add":{"path":"20230420_221059_14321_vy3rm-4e793728-5754-45ad-9e18-3
d6e48d995fb","partitionValues":{},"size":701,"modificationTime":16820
28661018,"dataChange":true,"stats":"{\"numRecords\":3,\"minValues\":{
\"id\":2,\"name\":\"FOUR\",\"description\":\"UPPER-CASED name col via
1st UPDATE
statement\"},\"maxValues\":{\"id\":6,\"name\":\"TWO\",\"description\"
:\"UPPER-CASED name col via 1st UPDATE
statement\"},\"nullCount\":{\"id\":0,\"name\":0,\"description\":0}}",
"tags":{}}}
```


E - Identify replacement file to be added


```
{"add":{"path":"20230420_221059_14321_vy3rm_671333dd-9340-47b2-8e5f-e
87b03e578fe","partitionValues":{},"size":617,"modificationTime":16820
28661112,"dataChange":true,"stats":"{\"numRecords\":2,\"minValues\":{
\"id\":1,\"name\":\"one\",\"description\":\"added via 1st INSERT
statement\"},\"maxValues\":{\"id\":3,\"name\":\"three\",\"description
\":\"added via 1st INSERT
statement\"},\"nullCount\":{\"id\":0,\"name\":0,\"description\":0}}",
"tags":{}}}
```


F - Identify replacement file to be added


```
{"add":{"path":"20230420_221059_14321_vy3rm_588d8b2b-2591-4a1c-98fc-9
7529f9ad92a","partitionValues":{},"size":591,"modificationTime":16820
28661305,"dataChange":true,"stats":"{\"numRecords\":1,\"minValues\":{
\"id\":5,\"name\":\"five\",\"description\":\"added via 2nd INSERT
statement\"},\"maxValues\":{\"id\":5,\"name\":\"five\",\"description\
":\"added via 2nd INSERT
statement\"},\"nullCount\":{\"id\":0,\"name\":0,\"description\":0}}",
"tags":{}}}
```


Lastly, lines **D** , **E**, and **F** identify that replacement files will be added to replace those being deleted. These new files will reflect the changes made to the table, and their addition will complete the update process.

Step 10 - Understanding Deltas

After running the next statement, what do you think might happen in S3?

```
INSERT INTO my_delta_tbl
 SELECT * FROM my_delta_tbl;
```

Yes, there is a new log file that identifies one or more new data files being created, in addition to the data files already making up the contents of the table. In the table's base folder, there will be the exact 1+ new files identified in the log file.

Again, remember that you are not expected to fully understand the exact mechanisms used by Delta Lake at this time. As the name suggests, it maintains a log of changes or **Deltas** on the underlying data **lake** files. This is the process that gives **Delta Lake** its name.

Step 11 - Understanding how Delta Log tracks deletions

This general practice of maintaining deltas is the common approach with modern table formats. To prevent this approach from becoming unmanageable when the `INSERTS`, `UPDATES`, and `DELETES` are coming in at a high rate, each of the table formats that support data modifications have its own "compaction" process that rolls changes together into fewer numbers of larger files and truncate the logs.

To learn more about how Delta Lake ultimately handles these logical steps, please consult the documentation.

Make sure deletions work with the table created by removing the records that have odd `id` values.

```
DELETE FROM
 my_delta_tbl
WHERE
 (id % 2) = 1;
```

```
28  SELECT * FROM students.lestertx.my_delta_tbl ORDER BY id;
29
```

Finished | Avg. read speed: 3.2 rows/s | Elapsed time: 1s | Rows: 6 | Query details

id	name	description
2	TWO	UPPER-CASED name col via 1st UPDATE statement
2	TWO	UPPER-CASED name col via 1st UPDATE statement
4	FOUR	UPPER-CASED name col via 1st UPDATE statement
4	FOUR	UPPER-CASED name col via 1st UPDATE statement
6	SIX	UPPER-CASED name col via 1st UPDATE statement
6	SIX	UPPER-CASED name col via 1st UPDATE statement

END OF LAB EXERCISE

Apache Iceberg

Lab 1: Create and populate Iceberg tables

Estimated completion time

- 30 minutes

Learning objectives

- In this lab, you will be introduced to the Iceberg table format and its many advantages. You will learn how to create Iceberg tables using Parquet and other file types, and explore how to create tables, add records, download results, and query values using Iceberg. You will also explore how conditional logic can be used to return precise results and how this is important for efficiency.

Prerequisites

- Student setup - Lab 1: Create student account

Activities

1. Prepare for the exercise
2. Create Iceberg tables using Parquet
3. Research Iceberg connector documentation
4. Create Iceberg table without defining format
5. Add records to Iceberg table
6. Simplify file list results
7. Find distinct values
8. Review documentation and list files
9. Add additional records to table
10. Add records using logical conditions

Step 1 - Prepare for the exercise

- Sign in and verify the `students` role is selected in the upper-right corner.
- Ensure the `aws-us-east-1-free` **Cluster** is reporting a **Status** of `Running`.
- In the **Query editor**, select `aws-us-east-1-free` in the cluster pulldown.

Note: *If you did not previously create a schema*, execute the following SQL statement. Make sure that you replace `yourname` with your actual name, or another identifier that you prefer. **Only use lowercase characters and numbers; no special characters or spaces.**

```
CREATE SCHEMA students.yourname;
```


Step 2 - Create Iceberg tables using Parquet

Create a new table in your schema named `my_iceberg_tbl` as shown below. Notice the `WITH` clause ensuring that it is created with the Iceberg type.

```
USE students.yourname;


CREATE TABLE my_iceberg_tbl (
   id integer,
   name varchar(55),
   description varchar(255)
) WITH (
   TYPE = 'iceberg', FORMAT = 'parquet'
);
```

Step 3 - Research Iceberg connector documentation

Reference the Iceberg connector — Starburst Enterprise metadata table documentation for this exercise. You will use several of these tables in this lab, and you can also explore others on your own that may not be utilized in these instructions.

Note: You will have to reference tables by their name only, not by their fully qualified 3-part `catalog.schema.table` name. Leverage the `USE catalog.schema` command or select appropriate values from the **Select catalog** and **Select schema** pulldowns in the upper-right of the UI.

Verify the file format type for this table.

```
SELECT * FROM "my_iceberg_tbl$properties";
```

Step 4 - Create Iceberg table without defining format

Create another table like the last one, but do not provide a file format in the `WITH` clause.

```
CREATE TABLE my_iceberg_tbl_dff (
  id integer,
  name varchar(55),
  description varchar(255)
) WITH ( TYPE = 'iceberg' );
```

See what the `$properties` for this new table report was used for the default file format.

```
SELECT * FROM "my_iceberg_tbl_dff$properties";
```

key	value
write.format.default	PARQUET

As shown above, the Parquet file format is the default file format.

Step 5 - Add records to Iceberg table

Add three new records to the table using the code below.

```
INSERT INTO my_iceberg_tbl
 (id, name, description)
VALUES
 (101, 'Leto', 'Ruler of House Atreides'),
 (102, 'Jessica', 'Concubine of the Duke'),
 (103, 'Paul', 'Son of Leto (aka Dale Cooper)');
```

Find out what file(s) these records are stored in.

```
SELECT
 name,
 "$path"
FROM
 my_iceberg_tbl;
```

How many files are present? If it is hard to tell from the UI due to the long values for the `$path` special column, you could perform one or more of the following steps to help yourself.

Step 6 - Simply file list results

Trim down the value to only pick up after `/data/` is found in the full path using the code below.

```
SELECT
 name,
 substring("$path", position('/data/' IN "$path") + 6)
FROM
 my_iceberg_tbl;
```

name	_col1
Leto	20230420_224449_31559_vy3rm-0275197c-9c55-45fe-b826-a78023a9f7ee.parquet
Jessica	20230420_224449_31559_vy3rm-0275197c-9c55-45fe-b826-a78023a9f7ee.parquet
Paul	20230420_224449_31559_vy3rm-0275197c-9c55-45fe-b826-a78023a9f7ee.parquet

With a quick look, it should appear that all three of these records are referencing the same file on the data lake.

Step 7 - Find distinct values

Use the code below to return a list of distinct values.

```
SELECT
 DISTINCT(substring("$path", position('/data/' IN "$path") + 6))
FROM
 my_iceberg_tbl;
```

Verify you have a single Parquet file for the three records inserted.

_col0
20230420_224449_31559_vy3rm-0275197c-9c55-45fe-b826-a78023a9f7ee.parquet

Step 8 - Review documentation and list files

Starburst Documentation outlines several ways to list the files corresponding to all of the rows in a table.

Review the material in the Iceberg connector — Starburst Enterprise section, and then query your table using the code below.

```
SELECT * FROM "my_iceberg_tbl$files";
```

You should see a single file – the same one from **Step 7**. Return specific fields for deeper review.

```
SELECT
 substring(file_path, position('/data/' IN file_path) + 6),
 record_count,
 value_counts,
 null_value_counts,
 lower_bounds,
 upper_bounds
FROM
 "my_iceberg_tbl$files";
```

Column	Value	Notes
file_path	20221224...*shortened*...f6e3.parquet	A specific file name. The remainder of the columns relate to this particular file
record_count	3	There are 3 records in the file
value_counts	{ 1 = 3, 2 = 3, 3 = 3 }	Each of the 3 columns has 3 values
null_value_counts	{ 1 = 0, 2 = 0, 3 = 0 }	None of the columns have any null values present
lower_bounds	{ 1 = 101, 2 = Jessica, 3 = Concubine of the }	> The id field ranges from 101 to 103 > The name field ranges from Jessica to Paul > The description field ranges from Concubine… to Son…
upper_bounds	{ 1 = 103, 2 = Paul, 3 = Son of Leto }	

Step 9 - Add additional records to table

Add a few more rows.

```
INSERT INTO my_iceberg_tbl
 (id, name, description)
VALUES
 (104, 'Thufir', 'Mentat'),
 (201, 'Vladimir', 'Ruler of House Harkonnen'),
 (202, 'Rabban', 'Ruthless nephew of Vladimir'),
 (203, 'Feyd-Rautha', 'Savvy nephew of Vladimir (played by Sting)'),
 (301, 'Reverend Mother Gaius Helen Mohiam', null);
```

Run the `$files` query again from **Step 8** and review your findings from the new data file just created.

Column	Value	Notes
`file_path`	`20221224...shortened... g2m3e.parquet`	A specific file name. The remainder of the columns relate to this particular file
`record_count`	`5`	There are 3 records in the file
`value_counts`	`{ 1 = 5, 2 = 5, 3 = 5 }`	Each of the 3 columns has 5 values
`null_value_counts`	`{ 1 = 0, 2 = 0, 3 = 1 }`	One of the rows has a null for the `description` field
`lower_bounds`	`{ 1 = 104, 2 = Feyd-Rautha, 3 = Mentat }`	> The `id` field ranges from `104` to `301` > The `name` field ranges from `Feyd-Rautha` to `Vladimir` > The `description` field ranges from `Mentat` to `Savy nephew…`
`upper_bounds`	`{ 1 = 301, 2 = Vladimir, 3 = Savvy nephew of! }`	

END OF LAB EXERCISE

Lab 2: Explore partitions with Iceberg

Estimated completion time

- 30 minutes

Learning objectives

- In this lesson, you will explore partitioning with Iceberg tables using a public bicycle-sharing dataset known as Bluebikes. You will learn how to access these tables and partition them in different ways. You will also compare how the Hive table format differs from the Iceberg table format, deploying a series of tests and checking the results.

Prerequisites

- Student setup - Lab 1: Create student account

Activities

1. Prepare for the exercise
2. Review Bluebikes dataset
3. Inspect column types
4. Count number of stations in each district
5. Partition table by district
6. Reorder columns to fit Hive requirements
7. Partition Iceberg table without changing order of columns
8. Copy non-partitioned table data into Iceberg table
9. Investigate files
10. Query partitions table
11. Reformat output
12. Query trips table
13. Partitioning start_time field
14. Understanding conditional logic and the `WHERE` clause
15. Create partitioned table
16. Copy data into table
17. Apply conditional logic using `WHERE` clause on source table
18. Compare results with newly partitioned table

Step 1 - Prepare for the exercise

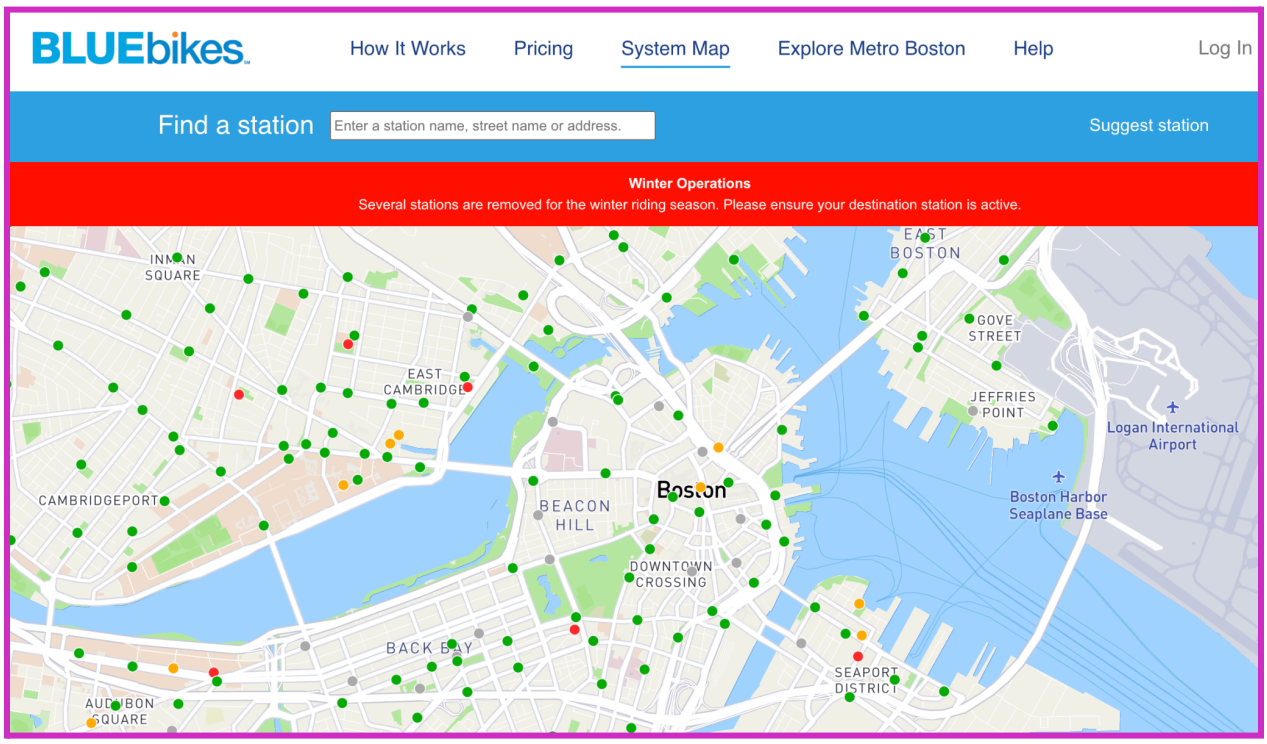
- Sign in and verify the `students` role is selected in the upper-right corner.
- Ensure the `aws-us-east-1-free` **Cluster** is reporting a **Status** of `Running`.
- In the **Query editor**, select `aws-us-east-1-free` in the cluster pulldown.

Note: *If you did not previously create a schema*, execute the following SQL statement. Make sure that you replace `yourname` with your actual name, or another identifier that you prefer. **Only use lowercase characters and numbers; no special characters or spaces.**

```
CREATE SCHEMA students.yourname;
```

Step 2 - Review Bluebikes dataset

For this lab, you will use the publicly available Bluebikes - Hubway dataset. Read more information about Blue Bikes Boston, a bicycle-sharing program based in Boston since 2011.



The two tables you will use are:

- `lakehouse.bluebikes.stations`
  - A list of the current docking stations available
  - Data current as of December 5, 2022
- `lakehouse.bluebikes.trips`
  - Transactional records of bike trips from start to finish
  - Contains all rides from January - November of 2022

Note: There are tables present whose names begin with `raw_` as well. The two tables above are examples of transformed/enriched/augmented/cleansed/etc. data that will be explored later in the data pipelining workshop.

Step 3 - Inspect column types

Using the `aws-us-east-1-free` cluster, familiarize yourself with these two tables by inspecting the column types and running a few queries to see some sample data. More details on specific columns can be found at Bluebikes System Data | Blue Bikes Boston.

Step 4 - Count number of stations in each district

Determine how many `stations` are assigned to each `district`. To do this, use a `SELECT` statement in conjunction with the `count()`, `GROUP BY`, and `ORDER BY` modifiers to introduce the desired logic.

```
SELECT
 district,
 count() as nbr_stations
FROM
 lakehouse.bluebikes.stations
GROUP BY
 district
ORDER BY
 nbr_stations DESC;
```

Step 5 - Partition table by district

Run the following code to see the full `CREATE TABLE` statement for the `stations` table.

```
SHOW CREATE TABLE lakehouse.bluebikes.stations;
```

Modify the output to create a new table partitioned by `district` in this first example. Notice how the SQL below specifically calls for a partitioning strategy by district.

```
USE students.yourname;

CREATE TABLE stations_orc_partby_district (
  station_nbr varchar,
  name varchar,
  latitude decimal(8, 6),
  longitude decimal(9, 6),
  district varchar,
  public boolean,
  total_docks smallint,
  deployment_year smallint
)
WITH (
  format = 'ORC', partitioned_by = ARRAY['district'], type = 'HIVE'
);
```

You should receive an error message similar to the image below. In this case, the partition keys must be the last column in the table, and listed in the same order as the table properties. Since this is not the case, the system generates an error.

Step 6 - Reorder columns to fit Hive requirements

The Hive table format is very particular about the placement of the partition column(s). To fix this, put it as the last column in the list using the code below.

```
CREATE TABLE stations_orc_partby_district (
  station_nbr varchar,
  name varchar,
  latitude decimal (8, 6),
  longitude decimal (9, 6),
  public boolean,
  total_docks smallint,
  deployment_year smallint,
  district varchar
)
WITH
(
  format = 'ORC', partitioned_by = ARRAY['district'], type = 'HIVE'
);
```


Note: You can populate this new table for investigation, but it is not necessary for the remainder of the exercise. The point to remember is the Hive table format requires the partition column(s) to be the last column(s) and in the same order.

Step 7 - Partition Iceberg table without changing order of columns

Use the following Data Definition Language (DDL) to create a partitioned Iceberg table. Note that the code below will partition the column **without** changing their order.

```
CREATE TABLE stations_p_district (
   station_nbr varchar,
   name varchar,
   latitude decimal (8, 6),
   longitude decimal (9, 6),
   district varchar,
   public boolean,
   total_docks smallint,
   deployment_year smallint
 )
WITH
 (
   type = 'iceberg',
   format = 'ORC',
   partitioning = ARRAY['district']
 );
```

The error that results is caused by subtle differences in supported data types between table formats. See Iceberg type mapping for more information.

Step 8 - Copy non-partitioned table data into Iceberg table

Review the integer data type family documentation and see if you can just go up to a wider data type. **Hint**: use `integer` instead as shown in the revised DDL.

```
CREATE TABLE stations_p_district (
  station_nbr varchar,
  name varchar,
  latitude decimal (8, 6),
  longitude decimal (9, 6),
  district varchar,
  public boolean,
  total_docks integer,
  deployment_year integer
)
WITH
(
  type = 'iceberg',
  format = 'ORC',
  partitioning = ARRAY['district']
);
```

Now copy the non-partitioned table's data into the partitioned Iceberg table you created.

```
INSERT INTO
 stations_p_district
SELECT
 *
FROM
 lakehouse.bluebikes.stations;
```

Step 9 - Investigate files

Take a look at the underlying files that were created using the code below.

```
SELECT
 substring(file_path, position('/data/' IN file_path) + 6),
 record_count
FROM
 "stations_p_district$files";
```

You were returned 14 rows as shown below. When you look closely you will see that the start of each of the file names begin with `district=`. These are the partitions that were created

for the 14 unique `distinct` values you determined in the count of stations by district query you ran in **Step 4**. Additionally, the totals from that step are also referenced appropriately in the `record_count` column values below.

_col0	record_count
district=Brookline/20230420_232957_44063_vy3rm-f8b7c386-bd50-42ad-8484-f1f5a85401b9.orc	14
district=Watertown/20230420_232957_44063_vy3rm-2550dab5-8195-4337-a938-5ba03e96ac6f.orc	6
district=Boston/20230420_232957_44063_vy3rm-1b3a9aae-d597-4d7e-8172-a2779b0633f7.orc	250
district=Somerville/20230420_232957_44063_vy3rm-76c5cd96-a81c-455d-91e3-d40bb4b51e68.orc	32
district=Cambridge/20230420_232957_44063_vy3rm-cd77bb9f-27f0-4c1a-be03-6efc031e40ae.orc	78
district=null/20230420_232957_44063_vy3rm-73ed61d7-6145-4871-aca3-820df7aa90b4.orc	4
district=Everett/20230420_232957_44063_vy3rm-0f23996a-1f07-4252-a1cc-dacd4e32f193.orc	14
district=Chelsea/20230420_232957_44063_vy3rm-395a69bb-6695-438c-950e-57724cfc23fd.orc	6
district=Newton/20230420_232957_44063_vy3rm-87b23299-0302-4cf1-829c-8078dd52883a.orc	15
district=Arlington/20230420_232957_44063_vy3rm-a146d88e-6bc1-4ea1-8c96-c55f82164f26.orc	6
district=Salem/20230420_232957_44063_vy3rm-3369571a-8b9e-4247-b917-f1d3831986f6.orc	14
district=Medford/20230420_232957_44063_vy3rm-3c5306bc-4876-48a1-9ff1-760125c8ea66.orc	3
district=Malden/20230420_232957_44063_vy3rm-2670fabf-6375-4671-a500-26164f5d7b96.orc	2
district=Revere/20230420_232957_44063_vy3rm-1ca91998-26db-4427-99ef-807697954d48.orc	4

You should also notice that the partition directory naming scheme is similar to that of the Hive table format. They both use the `name=value` format for the subdirectories.

Step 10 - Query partitions table

Review the Iceberg metadata tables. Iceberg tables include a `$partitions` table.

Use the code listed in the image below to query the `$partitions` table.

Note: The total number of records for each partition will be easier to obtain from this query than the prior one when we have more than a single file per partition.

```
SELECT * FROM "stations_p_district$partitions";
```

```
91  SELECT * FROM "stations_p_district$partitions";
```

Finished | Avg. read speed: 23.2 rows/s | Elapsed time: 0.60s | Rows: 14

partition	record_count	file_count	total_size
{ district = Salem }	14	1	1771
{ district = Newton }	15	1	1620
{ district = Cambridg...	78	1	3579
{ district = Everett }	14	1	1611
{ district = Chelsea }	6	1	1348
{ district = Watertow...	6	1	1377

Now look at the `data` column. It represents the column-level statistics across all the data files in a given partition folder.

```
{ station_nbr = { min = H32001, max = H32006, null_count = 0, nan_count =
NULL }, name = { min = 855 Broadway, max = Chelsea Station, null_count = 0,
nan_count = NULL }, latitude = { min = 42.390227, max = 42.398418, null_count
= 0, nan_count = NULL }, longitude = { min = -71.039872, max = -71.023909,
null_count = 0, nan_count = NULL }, public = { min = true, max = true,
null_count = 0, nan_count = NULL }, total_docks = { min = 11, max = 12,
null_count = 0, nan_count = NULL }, deployment_year = { min = 2021, max =
2021, null_count = 0, nan_count = NULL } }
```

Step 11 - Reformat output

With some simple formatting, it is easier to consume the column-level metadata for a particular partition.

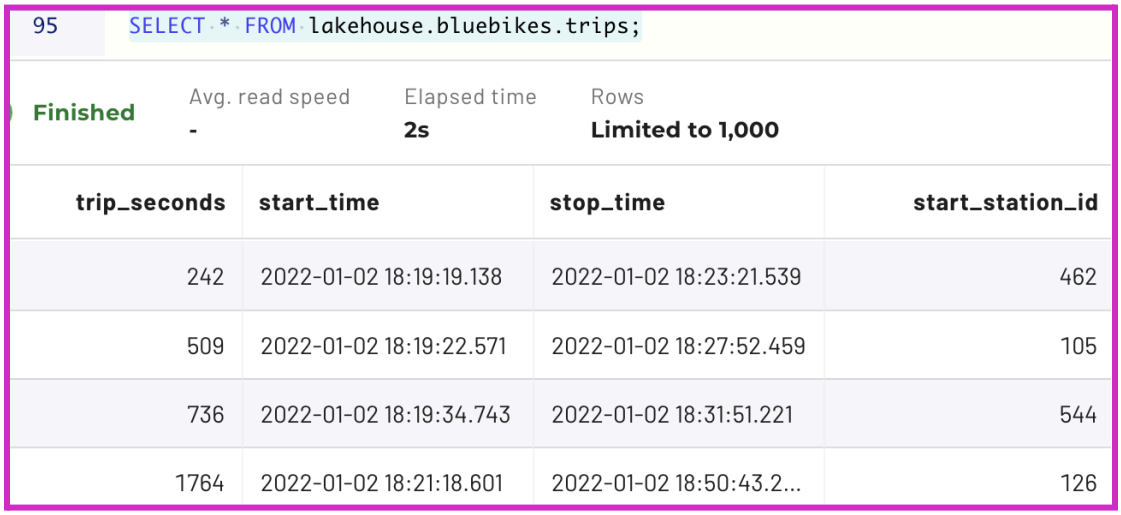
```
{
  station_nbr = { min = H32001, max = H32006, null_count = 0, nan_count = NULL },
  name = { min = 855 Broadway, max = Chelsea Station, null_count = 0, nan_count = NULL },
  latitude = { min = 42.390227, max = 42.398418, null_count = 0, nan_count = NULL },
  longitude = { min = -71.039872, max = -71.023909, null_count = 0, nan_count = NULL },
  public = { min = true, max = true, null_count = 0, nan_count = NULL },
  total_docks = { min = 11, max = 12, null_count = 0, nan_count = NULL },
  deployment_year = { min = 2021, max = 2021, null_count = 0, nan_count = NULL }
}
```

Since the partitioning was defined on the `district` column, Iceberg does not need to store any column statistics for that particular column. Mainly because there is only a single value, the `district` value, for the whole partition. The example above was from `district=Chelsea`.

Step 12 - Query trips table

It's time to turn your attention to the `lakehouse.bluebikes.trips` table. `SELECT` all records from the table using the code below.

```
SELECT * FROM lakehouse.bluebikes.trips;
```



trip_seconds	start_time	stop_time	start_station_id
242	2022-01-02 18:19:19.138	2022-01-02 18:23:21.539	462
509	2022-01-02 18:19:22.571	2022-01-02 18:27:52.459	105
736	2022-01-02 18:19:34.743	2022-01-02 18:31:51.221	544
1764	2022-01-02 18:21:18.601	2022-01-02 18:50:43.2...	126

Notice that the timestamp fields listed correspond to the start of each ride.

Step 13 - Partitioning start_time field

You know that the `start_time` data is immutable. You also know that many analytic queries include a filter targeting trips within a range of 1-180 days.

Based on these facts, it would be a good idea to consider partitioning based on the `start_time` field. Importantly, you cannot use discrete values to do this. This approach would be too granular and create too many folders containing only a single trip, introducing the small files problem.

Step 14 - Understanding conditional logic and the `WHERE` clause

Adopting a partitioning strategy based on month would be best in this case. The Hive table format allows you to abstract the level of precision from the `start_time` field and put that value in a new, partitioned column. The result will be a `start_time_month` field with a value format of `YYYY-MM`.

To use this approach efficiently, you will need to understand how these two columns are best utilized, balancing precision against efficiency. The goal is to target the smallest amount of data possible while still achieving a good result.

More directly, the Hive table format often causes confusion in scenarios like this by having two different columns (`start_time` and `start_time_month`) representing the same information.

Step 15 - Create partitioned table

Iceberg offers a much simpler solution to solve the problem in the previous step. The approach in the DDL below shows there are timestamp-based transform functions that can be used that do not require you to actually create & populate another dedicated column to achieve partitioning at a more coarse-grained level such as at the month level.

The user then does not have the opportunity to get confused about having two similar, but functionally different, date/timestamp fields. The user just applies the appropriate `WHERE` clause on the timestamp field.

Create a new table, partitioned by month using the code below.

```
CREATE TABLE trips_p_start_month (
  trip_seconds integer,
  start_time timestamp(6),
  stop_time timestamp(6),
  start_station_id integer,
  start_station_name varchar,
  start_station_latitude decimal(15, 13),
  start_station_longitude decimal(16, 13),
  end_station_id integer,
  end_station_name varchar,
  end_station_latitude decimal(15, 13),
  end_station_longitude decimal(16, 13),
  bike_id integer,
  user_type varchar,
  postal_code varchar
)
WITH (
  type = 'iceberg',
  format = 'ORC',
  partitioning = ARRAY['month(start_time)']
);
```

Step 16 - Copy data into table

Now it's time to copy the original data into your new table. Use the code below:


```
INSERT INTO trips_p_start_month
SELECT * FROM lakehouse.bluebikes.trips;
```

Step 17 - Apply conditional logic using `WHERE` clause on source table

From the source table from the prior query, `lakehouse.bluebikes.trips`, determine how many trips were started in March 2022.

```
SELECT COUNT(*) FROM lakehouse.bluebikes.trips
 WHERE start_time >=
        from_iso8601_timestamp('2022-03-01T00:00:00')
  AND start_time <
        from_iso8601_timestamp('2022-04-01T00:00:00');
```

You will see there are 182,421 results from that time range on the original table.

Step 18 - Compare results with newly partitioned table

Compare those results from what is seen in the `$partitions` metadata table. Use the code listed below.

```
SELECT * FROM "trips_p_start_month$partitions";
```

You can see that the `start_time_month=` partitions are not as straightforward as the purpose-built ones that you saw previously. The documentation notes the following:

Note: A partition is created for each month of each year. The partition value is the integer difference in months between the timestamp and January 1, 1970.

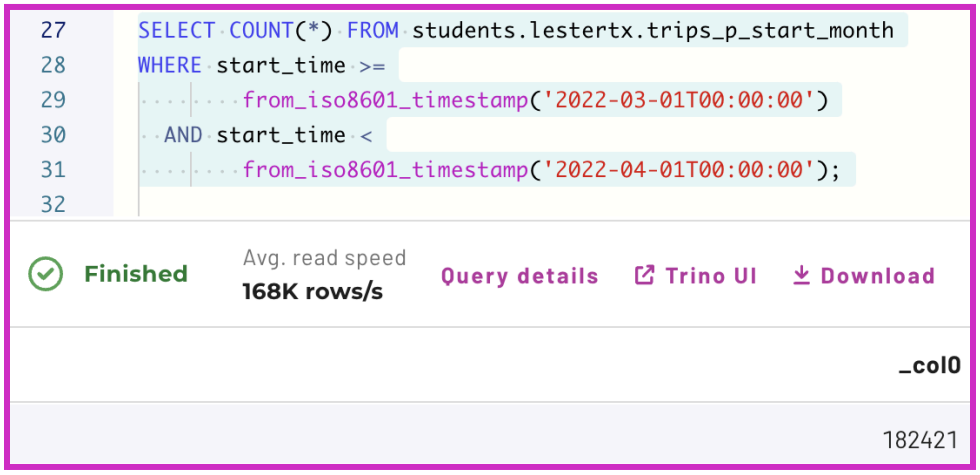
January 1, 1970, is the date known as the Unix Epoch, which is used as the start date for many systems, including this one. In your case, there are 52 years and 2 months before March 1, 2022 and the Unix Epoch, so the number you are looking for is 626. *(12 * 52 + 2 = 626).*

partition	record_count
{ start_time_month = 626 }	182421

This shows the metadata table identifying the correct number of records for the month of March 2022 from the appropriate partition of the newly partitioned table that we saw from the source table in **Step 17**.

Run the query again, this time on the partitioned table, to make sure the same 182,421 count is being returned.

```
SELECT COUNT(*) FROM trips_p_start_month
WHERE start_time >=
       from_iso8601_timestamp('2022-03-01T00:00:00')
 AND start_time <
       from_iso8601_timestamp('2022-04-01T00:00:00');
```



END OF LAB EXERCISE

Lab 3: Data modifications and snapshots with Iceberg

Estimated completion time

- 30 minutes

Learning objectives

- In this lesson, you will explore how Iceberg uses snapshot files to create a comprehensive picture of changes made to tables. To do this, you will set up a table, make changes, then view the impact that this has on snapshot files. You will then query the snapshot files using the Iceberg time travel feature, which allows you to view results from the database at various moments in time or even roll the table back to a previous state.

Prerequisites

- Student setup - Lab 1: Create student account

Activities

1. Prepare for the exercise
2. Create testing table
3. Add historical order records from a week ago
4. Add activation records from six days ago
5. Add an error record from five days ago
6. Modify some previous records
7. Understanding snapshot files
8. Time travel queries with snapshot id
9. Time travel queries against a point in time
10. Rolling back to a previous state using snapshot files

Step 1 - Prepare for the exercise

- Sign in and verify the `students` role is selected in the upper-right corner.
- Ensure the `aws-us-east-1-free` **Cluster** is reporting a **Status** of `Running`.
- In the **Query editor**, select `aws-us-east-1-free` in the cluster pulldown.

Note: *If you did not previously create a schema*, execute the following SQL statement. Make sure that you replace `yourname` with your actual name, or another identifier that you prefer. **Only use lowercase characters and numbers; no special characters or spaces.**

```
CREATE SCHEMA students.yourname;
```

Step 2 - Create testing table

Create a new table in your schema with the following Data Definition Language (DDL).

```
USE students.yourname;

CREATE TABLE phone_provisioning (
 phone_nbr bigint,
 event_time timestamp(6),
 action varchar(15),
 notes varchar(150)
)
WITH (
 type='iceberg',
 partitioning=ARRAY['day(event_time)']
);
```

Verify that a snapshot was created for the table creation.

```
SELECT * FROM "phone_provisioning$snapshots";
```

This returns a single row representing the first snapshot which captures the table creation itself. The following is the value of the `summary` column.

```
{ changed-partition-count = 0, total-equality-deletes = 0, total-position-deletes = 0,
total-delete-files = 0, total-files-size = 0, total-records = 0, total-data-files = 0
}
```

This indicates that no data changes happened as part of this snapshot. In fact, it also shows there are zero `total-records` as you would expect having only run the DDL.

Step 3 - Add historical order records from a week ago

Add historical records from a week ago to capture the initial orders for two new phone numbers.

```
INSERT INTO
 phone_provisioning (phone_nbr, event_time, action, notes)
VALUES
 (
   1111111, current_timestamp(6) - interval '7' day, 'ordered', null
 ),
 (
   2222222, current_timestamp(6) - interval '7' day, 'ordered', null
 );
```

Verify the two records are present as expected.

```
SELECT * FROM phone_provisioning ORDER BY event_time DESC;
```

Verify a new snapshot was created.

```
SELECT * FROM "phone_provisioning$snapshots";
```

There is now a second row with a new `snapshot_id`. More analysis of this metadata will be discussed in **Step 7**.

committed_at	snapshot_id	parent_id	operation	manifest_list	summary
2023-04-21 09:53:3...	39760182185247010...	NULL	append	s3://edu-train-galax...	{ changed-parti
2023-04-21 09:57:4...	23934521210609399...	39760182185247010...	append	s3://edu-train-galax...	{ changed-parti

Step 4 - Add activation records from six days ago

Add historical records from six days ago to capture the activation activity for the same two phone numbers.

```
INSERT INTO
 phone_provisioning (phone_nbr, event_time, action, notes)
VALUES
 (
   1111111, current_timestamp(6) - interval '6' day, 'activated',
null
 ),
 (
   2222222, current_timestamp(6) - interval '6' day, 'activated',
null
 );
```

Verify the records are present as expected, and a new snapshot was created, by running the last two `SELECT` statements in **Step 3**. There will be two more records added to the `phone_provisioning` table and the `$snapshot` table will now have a third record.

Reviewing the `summary` column for the new record in `$snapshot` details that 2 records were added as part of this snapshot.

```
{ changed-partition-count = 1, added-data-files = 1, total-equality-
deletes = 0, added-records = 2, trino_query_id =
20230421_070553_71797_vy3rm, total-position-deletes = 0, added-files-
size = 742, total-delete-files = 0, total-files-size = 1468, total-
records = 4, total-data-files = 2 }
```

More analysis will be presented in **Step 7**.

Step 5 - Add an error record from five days ago

Add historical activation records from five days ago to capture an error that was reported on phone number `2222222`.

```
INSERT INTO
  phone_provisioning (phone_nbr, event_time, action, notes)
VALUES
  (2222222, current_timestamp(6) - interval '5' day, 'errorReported',
  'customer reports unable to initiate call');
```

Once again, verify the records are present as expected and a new snapshot was created.

Step 6 - Modify some previous records

Four days ago, a system error prevented the `notes` column from being populated correctly before it was fixed. Upon review of the problem, it was determined that the following two `INSERT` commands are needed to modify the affected records.

```
UPDATE phone_provisioning
  SET notes = 'customer requested new number'
WHERE action = 'ordered'
  AND notes is null;

UPDATE phone_provisioning
  SET notes = 'number successfully activated'
WHERE action = 'activated'
  AND notes is null;
```

Review all rows again.

```
SELECT * FROM phone_provisioning ORDER BY event_time DESC;
```

Your `event_time` values will be different, but the rows shown below should be similar to your results.

```
42  SELECT * FROM phone_provisioning ORDER BY event_time DESC;
43
```

Finished | Avg. read speed: 8 rows/s | Elapsed time: 0.63s | Rows: 5 | Query details

phone_nbr	event_time	action	notes
2222222	2022-12-22 12:43:55.286210	errorReported	customer reports unable to initiate call
1111111	2022-12-21 12:30:31.800953	activated	number successfully activated
2222222	2022-12-21 12:30:31.800953	activated	number successfully activated
2222222	2022-12-20 12:20:50.457304	ordered	customer requested new number
1111111	2022-12-20 12:20:50.457304	ordered	customer requested new number

Query the `$snapshots` table again.

```
SELECT * FROM "phone_provisioning$snapshots";
```

Two new snapshots were created from the two `UPDATE` statements, as shown at the bottom of the list below.

Finished | Avg. read speed: 22.1 rows/s | Elapsed time: 0.27s | Rows: 6 | Results from cache: No

committed_at	snapshot_id	parent_id	operation
2024-03-25 17:48:54.7...	791119850705384616	NULL	append
2024-03-25 17:49:08.9...	5833632576367314300	791119850705384616	append
2024-03-25 17:49:42.9...	7343058207057648992	5833632576367314300	append
2024-03-25 17:50:06.5...	3847696610631931623	7343058207057648992	append
2024-03-25 17:51:35.27...	4909038268719538118	3847696610631931623	overwrite
2024-03-25 17:53:11.73...	1526268464386354821	4909038268719538118	overwrite

Step 7 - Understanding snapshot files

Iceberg records changes to a table as snapshots. For example, each of the 2 `UPDATE` statements executed created their own snapshot with an `operation` value of `overwrite`. Iceberg cannot perform an in-place update to the underlying immutable data files.

The phrase "overwrite", instead of "update", is more appropriate as Iceberg has to create a file referencing the location in the existing file(s) that contain the record(s) to be updated. This is called a "delete file".

Then, as part of an atomic operation, Iceberg creates a new data file that has the full record being "updated". It essentially is an "add" of the record with all updated columns as well as the existing values for columns not updated.

These new delete files and data files that are created will be read after the preceding data files are. This allows Iceberg to modify the data prior to returning it by applying the delete files (i.e. delete the records) and then including the new data files (which will look like net-new records).

This is somewhat analogous to RDBMS "transaction logs" that store a running history of modifications into. The difference is that the classical databases are creating their transaction logs for recovery & replication purposes. Iceberg creates a series of deltas that will be used in a "merge on read" strategy when the table is queried.

To understand this better, click on the `summary` value for the last snapshot in the list. The **Output** identified below is a subset of all the properties present. They are the ones most important to this discussion.

Output

```
{
 total-position-deletes = 2,
 total-delete-files = 1,

 added-records = 2,
 added-data-files = 1,
}
```

These totals are for the `UPDATE` statement that had `action = 'activated' AND notes is null` within it. It logically changed two records.

With Iceberg's inability to perform in-place updates on the underlying files, `total-position-deletes = 2` indicates that 2 records were marked for deletion. Fortunately, both of these were placed in a single delete file.

Closely following those deletes, `added-records = 2` indicates the records deleted are being re-added as essentially new inserts. As before, these were assembled into a single new data file.

More information on delete files and the overall strategy around handling the `UPDATE` statement can be found in the Iceberg Specification.

Step 8 - Time travel queries with snapshot id

So, what can you do with snapshot files? One useful feature is called "time travel". This allows you to query prior versions of the table via the `snapshot_id`. The second row in the results below relates to the second snapshot. The first snapshot was at the empty table creation. The second was the initial `INSERT` statement.

```
45  SELECT * FROM "phone_provisioning$snapshots";
46
```

Finished | Avg. read speed: 8.3 rows/s | Elapsed time: 0.97s | Rows: 8 | Query details

committed_at	snapshot_id	parent_id	operation
2022-12-27 12:01:31.5...	5641615054948073642	NULL	append
2022-12-27 12:20:52....	2701885389952950484	5641615054948073642	append
2022-12-27 12:30:32...	4503567719363821411	2701885389952950484	append

Replace `12345678890` in the following query with your second `snapshot_id` to see what the table looked like back at that point.

```
SELECT * FROM phone_provisioning
FOR VERSION AS OF 1234567890
ORDER BY event_time DESC;
```

You should see the first two records initially added, similar to the image pictured below.

phone_nbr	event_time	action	notes
1111111	2022-12-20 12:20:50.457304	ordered	NULL
2222222	2022-12-20 12:20:50.457304	ordered	NULL

Step 9 - Time travel queries against a point in time

Another alternative mechanism to leverage time travel is to run a query based on a past point in time. You can exercise this by swapping `VERSION` above with `TIMESTAMP` and replacing the `snapshot_id` with a timestamp.

Familiarize yourself with the documentation on time travel. Many unique use-cases make this feature invaluable.

Previously, we added a record with a timestamp that was 5 days in the past. Run a query to see what that table's contents were 5.5 days ago.

```
SELECT * FROM phone_provisioning
FOR TIMESTAMP AS OF current_timestamp(6) - interval '132' hour
ORDER BY event_time DESC;
```

You should have received an error like the following.

Do you understand what happened here? The query above assumed that the `event_time` column's timestamp was being used, but it is only a timestamp-based column and is not directly related to the versioning information. The rows from the `$snapshots` metadata table have a `committed_at` timestamp which is leveraged when `FOR TIMESTAMP AS OF` is utilized.

Run the last query again after changing the `interval` type from `hour` to `minute` and plug in a single-digit number instead of `132`. Increment and/or decrement the number until you get the results you are looking for.

Hint: use an appropriate number of minutes that would make the timestamp slightly before the `committed_at` column value from `$snapshots` for the snapshot you are trying to read the data from.

Step 10 - Rolling back to a previous state using snapshot files

Time travel also provides the ability to roll a table back to a previous snapshot. Use the following steps to determine the current `snapshot_id` and ***copy it into your query editor*** for later use.

To begin, query `$snapshots` again and get the `snapshot_id` value from the row with the most recent `committed_at` timestamp to be the snapshot to rollback to. You could also look for the most recent `snapshot_id` in the `$history` metadata table.

```
-- save the snapshot_id value in the editor
SELECT snapshot_id FROM "phone_provisioning$history"
ORDER BY made_current_at DESC LIMIT 1;
```

Next, execute a `SELECT *` query using the specific `snapshot_id` just identified to ensure the results are what are present where running a query **without** the `FOR VERSION AS OF` clause (replace 1234567890 with your current `snapshot_id`).

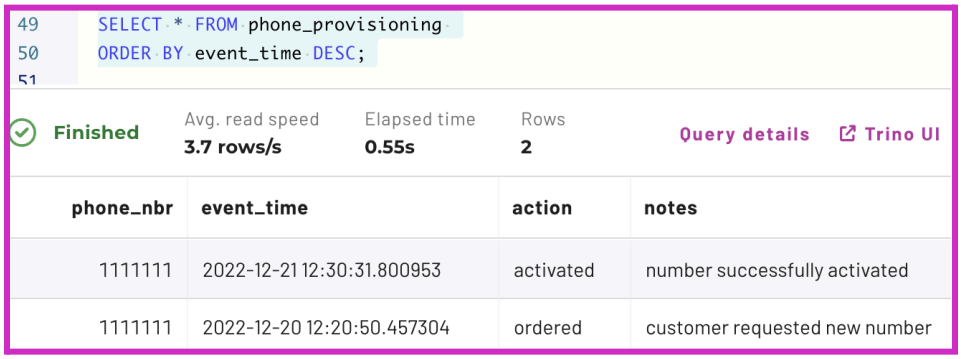
```
SELECT * FROM phone_provisioning
EXCEPT
SELECT * FROM phone_provisioning
FOR VERSION AS OF 1234567890;
```

Run the following query to remove all the records for one of the phone numbers.

```
DELETE FROM phone_provisioning
 WHERE phone_nbr = 2222222;
```

Verify that 3 of the 5 rows were deleted.

```
SELECT * FROM phone_provisioning ORDER BY event_time DESC;
```



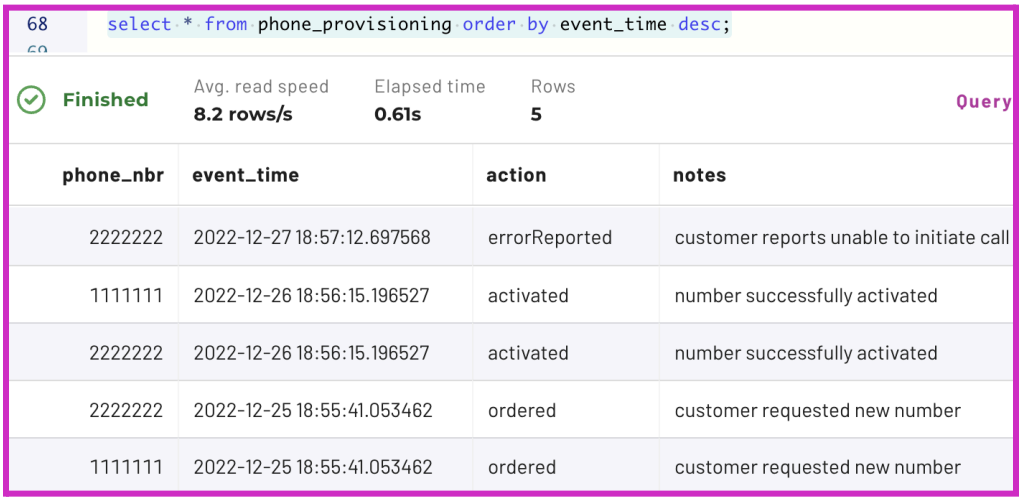
It was determined that the `DELETE` was a mistake and needs to be recovered from. Use the `snapshot_id` you saved earlier in the following command that rolls the table back to before the deletions.

```
CALL students.system.rollback_to_snapshot(
    'yourname', 'phone_provisioning',
     1234567890);
```

Finally, verify the rows have been recovered.

```
SELECT * FROM phone_provisioning ORDER BY event_time DESC;
```

The result should look similar to the image below.



phone_nbr	event_time	action	notes
2222222	2022-12-27 18:57:12.697568	errorReported	customer reports unable to initiate call
1111111	2022-12-26 18:56:15.196527	activated	number successfully activated
2222222	2022-12-26 18:56:15.196527	activated	number successfully activated
2222222	2022-12-25 18:55:41.053462	ordered	customer requested new number
1111111	2022-12-25 18:55:41.053462	ordered	customer requested new number

END OF LAB EXERCISE

Advanced Iceberg

Lab 1: Utilize Iceberg's `MERGE` statement

Estimated completion time

- 30 minutes

Learning objectives

- In this lesson, you will learn how to use the `MERGE` statement using Iceberg tables. To do this, you will create an Iceberg table, populate it with records, and record the behavior as changes are made. You will then explore the concept of Slowly Changing Dimensions (SCD) before using `UNION ALL` statements in conjunction with `MERGE` to enact a complete merge of Iceberg tables.

Prerequisites

- Student setup - Lab 1: Create student account

Activities

1. Prepare for the exercise
2. Create Iceberg table
3. Add records to table
4. Create table to record changes
5. Create changes upstream and view impact on records table
6. Merge change logs into table base
7. Understanding Slowly Changing Dimensions (SCD)
8. Populate records and mark as active
9. Understanding `MERGE`
10. Using the `UNION ALL` statement
11. Wrap UNION query with existing `MERGE` query
12. Mark records as inactive
13. Insert unmatched records and set as active
14. Run full `MERGE` command

Step 1 - Prepare for the exercise

- Sign in and verify the `students` role is selected in the upper-right corner.
- Ensure the `aws-us-east-1-free` **Cluster** is reporting a **Status** of `Running`.
- In the **Query editor**, select `aws-us-east-1-free` in the cluster pulldown.

Note: *If you did not previously create a schema*, execute the following SQL statement. Make sure that you replace `yourname` with your actual name, or another identifier that you prefer. **Only use lowercase characters and numbers; no special characters or spaces.**

```
CREATE SCHEMA students.yourname;
```

Step 2 - Create Iceberg table

Using the `aws-us-east-1-free` cluster, create a new table in your schema with the following Data Definition Language (DDL).

```
USE students.yourname;

CREATE TABLE cust_type1 (
  cust_id bigint,
  cust_name varchar(50),
  country varchar(50),
  vip_ind boolean
)
WITH (
  type='iceberg'
);
```

Step 3 - Add records to table

Load it with some initial records.

```
INSERT INTO cust_type1
 (cust_id, cust_name, country, vip_ind)
VALUES
 (101, 'Fred', 'United States', false),
 (102, 'Wilma', 'United States', false),
 (103, 'Pebbles', 'United States', false);
```

Step 4 - Create table to record changes

Create a table to hold the log-based changes being populated upstream.

```
CREATE TABLE cust_chg_log (
  cust_id bigint,
  cust_name varchar(50),
  country varchar(50),
  vip_ind boolean
)
WITH (
  type='iceberg'
);
```

Step 5 - Create changes upstream and view impact on records table

Simulate an upstream event that populates this table with a mix of new and updated records.

Note: this use case does not have an indicator to declare if the record is a change to an existing record or a new record.

```
INSERT INTO cust_chg_log
 (cust_id, cust_name, country, vip_ind)
VALUES
 (101, 'Fred', 'United States', true),
 (104, 'Betty', 'United States', true),
 (105, 'Barney', 'United States', false),
 (106, 'Bam-Bam', 'United States', false),
 (103, 'Pebbles', 'Canada', false);
```

From reviewing the current state of `cust_type1`, you can determine the first and last records are updates to `Fred` and `Pebbles` while the middle three are held in new records.

Step 6 - Merge change logs into table base

The `MERGE` statement begins with identifying you want to merge the change log records into the base table as shown below. Notice that it calls for the `cust_chg_log` records to be read and then merged into the `cust_type1` table.

```
MERGE INTO cust_type1 AS cust
USING cust_chg_log AS log
```

The next part is identifying the criteria used to determine a match. For our simple example, this is solely based on the `cust_id` values from both tables.

```
ON (cust.cust_id = log.cust_id)
```

When a match is found, an `UPDATE` needs to be performed to overwrite the entire record.

```
WHEN MATCHED THEN
UPDATE
 SET
   cust_id = log.cust_id,
   cust_name = log.cust_name,
   country = log.country,
   vip_ind = log.vip_ind
```

Conversely, when no match is made an `INSERT` needs to be executed to add the new record.

```
 WHEN NOT MATCHED THEN
  INSERT (cust_id, cust_name, country, vip_ind)
  VALUES
    (log.cust_id, log.cust_name, log.country,
 log.vip_ind);
```

Putting it all together, you have this completed statement to be executed.

```
MERGE INTO cust_type1 AS cust
USING cust_chg_log AS log
ON (cust.cust_id = log.cust_id)
WHEN MATCHED THEN
UPDATE
 SET
   cust_id = log.cust_id,
   cust_name = log.cust_name,
   country = log.country,
   vip_ind = log.vip_ind
WHEN NOT MATCHED THEN
 INSERT (cust_id, cust_name, country, vip_ind)
 VALUES (log.cust_id, log.cust_name, log.country, log.vip_ind);
```

After running the `MERGE` statement, review the contents of `cust_type1`.

```
SELECT * FROM cust_type1 ORDER BY cust_id;
```

Verify that the 3 new records were added as well as the modifications made to the `Fred` and `Pebbles` records. It should resemble the image below.

```
42  select * from cust_type1 order by cust_id;
43
```

Finished | Avg. read speed: 12.3 rows/s | Elapsed time: 0.49s | Rows: 6

cust_id	cust_name	country	vip_ind
101	Fred	United States	true
102	Wilma	United States	false
103	Pebbles	Canada	false
104	Betty	United States	true
105	Barney	United States	false
106	Bam-Bam	United States	false

Step 7 - Understanding Slowly Changing Dimensions (SCD)

This approach of overwriting the data when a change is detected is often referred to as a Slowly Changing Dimensions (SCD) Type 1 strategy. SCD Type 2 maintains history by updating the current record to indicate it is not the most current and creating a new record with the current values.

Note: Although there are six types, in practice, Type 1 and Type 2 are the most commonly used.

Run the code below to create a table that includes additional columns. This will allow for a Type 2 solution to be implemented.

```
CREATE TABLE cust_type2 (
  cust_id bigint,
  cust_name varchar(50),
  country varchar(50),
  vip_ind boolean,
  active_ind boolean,
  active_start_ts timestamp(6),
  active_stop_ts timestamp(6)
)
WITH (
  type='iceberg'
);
```

Step 8 - Populate records and mark as active

Populate the same initial three records as before, but this time, mark them active and set the date of this status beginning one month ago.

```
INSERT INTO cust_type2
 (cust_id, cust_name, country, vip_ind,
  active_ind, active_start_ts, active_stop_ts)
VALUES
 (101, 'Fred', 'United States', false,
  true, current_timestamp - INTERVAL '1' month, null),
 (102, 'Wilma', 'United States', false,
  true, current_timestamp - INTERVAL '1' month, null),
 (103, 'Pebbles', 'United States', false,
  true, current_timestamp - INTERVAL '1' month, null);
```

For this use case, you will still use the previously created `cust_chg_log` table with its simulated new and modified records.

Note: No additional information is needed. These records will become active when the `MERGE` statement executes.

Step 9 - Understanding `MERGE`

Review the documentation Supporting MERGE. Notice that only `UPDATE` and `DELETE` operations are possible for `WHEN MATCHED`. All `WHEN NOT MATCHED` cases only allow for `INSERT` operations. This approach works for all new records in the change log.

This presents a challenge in cases where a given record already has a change log present in the base table. In such cases, it is necessary to add the updated record, mark it as active, and mark the old record as inactive.

Step 10 - Using the `UNION ALL` statement

Let's unpack this process further. To get things ready for `MERGE`, you have to do a bit of groundwork to ensure that the change logs conform to the specifications above. To do this, you will use a new expression known as `UNION ALL`, which combines the result set of two or more `SELECT` statements.

Begin by constructing a Common Table Expression (CTE) using a `UNION ALL` statement.

The first query to be unioned adds an "is a match" column (set to null) to the `cust_chg_log` records. It also populates the additional `cust_type2` columns with appropriate values. This

takes all the rows in the change log and adds the additional columns we need to have to track for type 2.

```
SELECT null as cust_id_match, cust_id,
       cust_name, country, vip_ind,
       true as active_ind,
       current_timestamp as active_start_ts,
       null as active_stop_ts
  FROM cust_chg_log
```

The second query to be unioned identifies only records in the base table that need to be updated by those in the change log. This will confirm customers in the change log already exist in the customer table.

```
SELECT cust_id AS cust_id_match, cust_id,
       cust_name, country, vip_ind,
       active_ind,
       active_start_ts,
       active_stop_ts
  FROM cust_type2
 WHERE cust_id IN (SELECT cust_id FROM cust_chg_log)
```

Now it's time to pull it all together. Run the full `UNION` query as listed below. The results should look similar to the image beneath.

```
SELECT null as cust_id_match, cust_id,
       cust_name, country, vip_ind,
       true as active_ind,
       current_timestamp as active_start_ts,
       null as active_stop_ts
  FROM cust_chg_log
UNION ALL
SELECT cust_id AS cust_id_match, cust_id,
       cust_name, country, vip_ind,
       active_ind,
       active_start_ts,
       active_stop_ts
  FROM cust_type2
 WHERE cust_id IN (SELECT cust_id FROM cust_chg_log);
```

Finished | Avg. read speed 14.7 rows/s | Elapsed time 1s | Rows 7

cust_id_match	cust_id	cust_name	country	vip_ind	active_ind	active_start_ts	active_end_ts
NULL	101	Fred	United States	true	true	2023-01-02 02:20:29.4910...	NULL
NULL	104	Betty	United States	true	true	2023-01-02 02:20:29.4910...	NULL
NULL	105	Barney	United States	false	true	2023-01-02 02:20:29.4910...	NULL
NULL	106	Bam-Bam	United States	false	true	2023-01-02 02:20:29.4910...	NULL
NULL	103	Pebbles	Canada	false	true	2023-01-02 02:20:29.4910...	NULL
101	101	Fred	United States	false	true	2022-12-02 02:19:45.4540...	NULL
103	103	Pebbles	United States	false	true	2022-12-02 02:19:45.4540...	NULL

The results show two rows with values in the `cust_id_match` column. These are the existing `Fred` and `Pebbles` records that need to be marked inactive. The results also show 5 rows with null in the match field. These are the 5 rows from `cust_chg_log`.

- Two of them are the new records for `Fred` and `Pebbles` that represent their newly active values.
- The other three are the new records (`Barney`, `Betty`, and `Bam-Bam`) that need to be added to the base table.

Step 11 - Wrap `UNION` query with existing `MERGE` query

Wrap the `UNION` query above with the following SQL above and below your existing query. This allows you to find both records that already exist.

```
MERGE INTO cust_type2 AS cust
USING (
-- keep your UNION ALL query here
) AS scd_changes
ON (cust.cust_id = scd_changes.cust_id AND
    cust.cust_id = scd_changes.cust_id_match)
```

Step 12 - Mark records as inactive

Now mark these records as inactive.

```
WHEN MATCHED AND cust.active_ind = true THEN
 UPDATE SET active_ind = false,
            active_stop_ts = current_timestamp
```

Note: The additional check above takes into consideration that there may be more than one inactive record already existing.

Step 13 - Insert unmatched records and set as active

The unmatched records include both completely new records and new values for existing `cust_id` records. Insert them as active records using the code below.

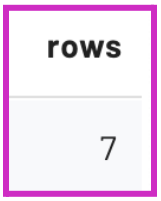
```
WHEN NOT MATCHED THEN
 INSERT (cust_id, cust_name, country,
         vip_ind, active_ind,
         active_start_ts, active_stop_ts)
 VALUES (scd_changes.cust_id,
         scd_changes.cust_name, scd_changes.country,
         scd_changes.vip_ind, scd_changes.active_ind,
         scd_changes.active_start_ts,
         scd_changes.active_stop_ts)
```

Step 14 - Run full `MERGE` command

Run the fully assembled `MERGE` command for this SCD Type 2 implementation.

```
MERGE INTO cust_type2 AS cust
USING (
SELECT null as cust_id_match, cust_id,
       cust_name, country, vip_ind,
       true as active_ind,
       current_timestamp as active_start_ts,
       null as active_stop_ts
  FROM cust_chg_log
UNION ALL
SELECT cust_id AS cust_id_match, cust_id,
       cust_name, country, vip_ind,
       active_ind,
       active_start_ts,
       active_stop_ts
  FROM cust_type2
 WHERE cust_id IN (SELECT cust_id FROM cust_chg_log)
) AS scd_changes
ON (cust.cust_id = scd_changes.cust_id AND
    cust.cust_id = scd_changes.cust_id_match)
WHEN MATCHED AND cust.active_ind = true THEN
 UPDATE SET active_ind = false,
            active_stop_ts = current_timestamp
WHEN NOT MATCHED THEN
 INSERT (cust_id, cust_name, country,
         vip_ind, active_ind,
         active_start_ts, active_stop_ts)
 VALUES (scd_changes.cust_id,
         scd_changes.cust_name, scd_changes.country,
         scd_changes.vip_ind, scd_changes.active_ind,
         scd_changes.active_start_ts,
         scd_changes.active_stop_ts);
```

Review the output. It should look similar to the image below, reporting seven updated rows.



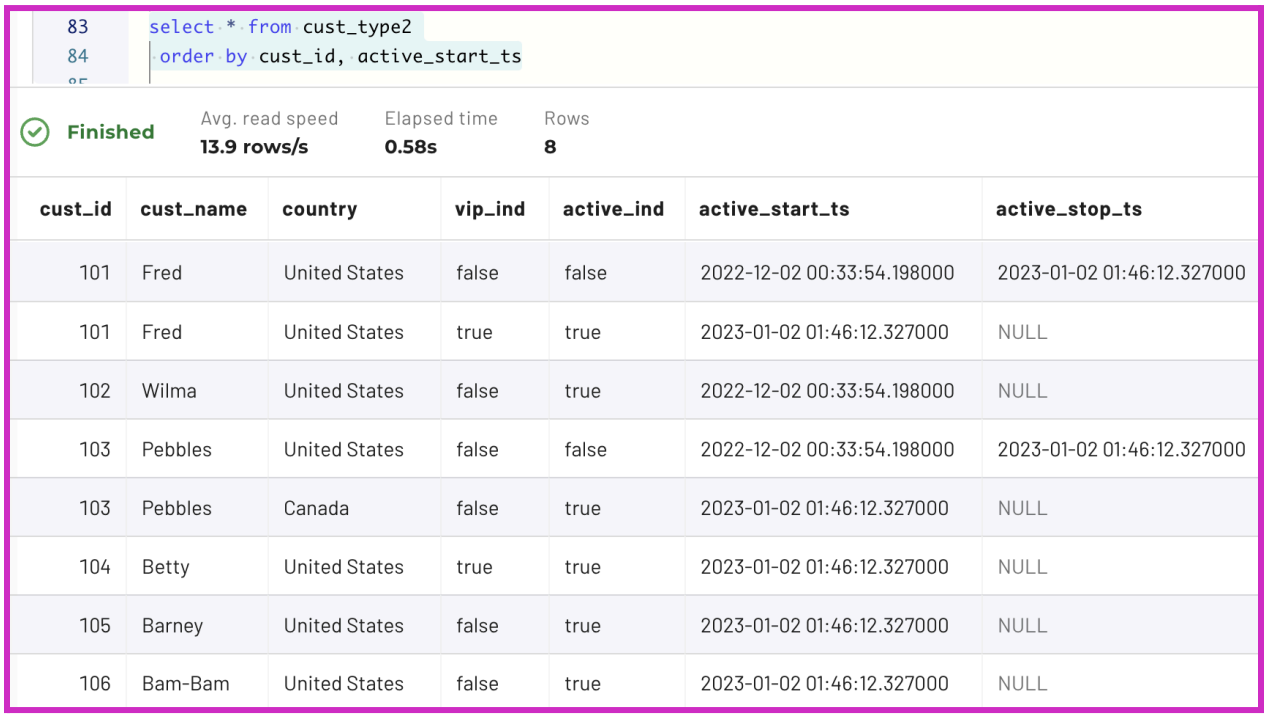
rows
7

Review the merged results.

```
SELECT * FROM cust_type2 ORDER BY cust_id, active_start_ts;
```

The table is correctly populated.

- Fred has been updated to now be a VIP and the original record has been marked inactive.
- Wilma still only shows the original record.
- Pebbles' country has been updated and the original record is marked inactive.
- The final 3 records are the completely new records from the change log.



```
83  select * from cust_type2
84  order by cust_id, active_start_ts
```

Finished | Avg. read speed: 13.9 rows/s | Elapsed time: 0.58s | Rows: 8

cust_id	cust_name	country	vip_ind	active_ind	active_start_ts	active_stop_ts
101	Fred	United States	false	false	2022-12-02 00:33:54.198000	2023-01-02 01:46:12.327000
101	Fred	United States	true	true	2023-01-02 01:46:12.327000	NULL
102	Wilma	United States	false	true	2022-12-02 00:33:54.198000	NULL
103	Pebbles	United States	false	false	2022-12-02 00:33:54.198000	2023-01-02 01:46:12.327000
103	Pebbles	Canada	false	true	2023-01-02 01:46:12.327000	NULL
104	Betty	United States	true	true	2023-01-02 01:46:12.327000	NULL
105	Barney	United States	false	true	2023-01-02 01:46:12.327000	NULL
106	Bam-Bam	United States	false	true	2023-01-02 01:46:12.327000	NULL

END OF LAB EXERCISE

Lab 2: Exercise advanced features of Iceberg

Estimated completion time

- 30 minutes

Learning objectives

- In this lab, you will explore Iceberg's advanced features using an airplane dataset. Specifically, you will learn how tables can evolve in specific ways, including renaming columns, adding additional columns, renaming partitions, and eliminating partitions. At each step, you will see what impact these changes have on the records in the table and investigate how these changes are tracked by Iceberg using snapshot files.

Prerequisites

- Student setup - Lab 1: Create student account

Activities

1. Prepare for the exercise
2. Create table
3. Insert records
4. Change column name using `ALTER TABLE`
5. Add new records
6. Add additional columns using `ALTER TABLE`
7. Modify existing records using new column layout
8. Rename and partition table
9. Add two new helicopters into modified table
10. Check partition metadata
11. Eliminate partitions
12. Add new record to table after partitions eliminated
13. Compaction

Step 1 - Prepare for the exercise

- Sign in and verify the `students` role is selected in the upper-right corner.
- Ensure the `aws-us-east-1-free` **Cluster** is reporting a **Status** of `Running`.
- In the **Query editor**, select `aws-us-east-1-free` in the cluster pulldown.

Note: *If you did not previously create a schema*, execute the following SQL statement. Make sure that you replace `yourname` with your actual name, or another identifier that you prefer. **Only use lowercase characters and numbers; no special characters or spaces.**

```
CREATE SCHEMA students.yourname;
```

Step 2 - Create table

In this lab, you will build and populate a table with airplane data.

Using the `aws-us-east-1-free` cluster, create an initial table layout in your schema with the following Data Definition Language (DDL).

```
USE students.yourname;

CREATE TABLE planes (
   tail_number varchar(15),
   name varchar(150),
   color varchar(15)
) WITH ( type = 'iceberg');
```

Step 3 - Insert records

Insert two new records into the table using the code below.

```
INSERT INTO planes (tail_number, name)
VALUES
 ('N707JT', 'John Travolta''s Boeing 707'),
 ('N1KE', 'Nike corp jet');
```

Step 4 - Change column name using `ALTER TABLE`

At this point, you realize the `name` is really more of a `description`. Change the column name using the code below. Note the use of the `ALTER TABLE` syntax.

```
ALTER TABLE planes RENAME COLUMN name TO description;
```

Verify all is reporting correctly.

```
SELECT * FROM planes;
```

Step 5 - Add new records

Add another plane.

```
INSERT INTO planes (tail_number, color, description)
VALUES
 ('N89TC', 'white',
  '1975 Bombardier Learjet 35 w/Light Jet classification');
```

Step 6 - Add additional columns using `ALTER TABLE`

At this point, you realize some additional columns are appropriate. Add them now using the code below.

```
ALTER TABLE planes ADD COLUMN class varchar(50);
ALTER TABLE planes ADD COLUMN year integer;
ALTER TABLE planes ADD COLUMN make varchar(100);
ALTER TABLE planes ADD COLUMN model varchar(100);
```

Additionally, the `color` column is not relevant enough to be maintained.

```
ALTER TABLE planes DROP COLUMN color;
```

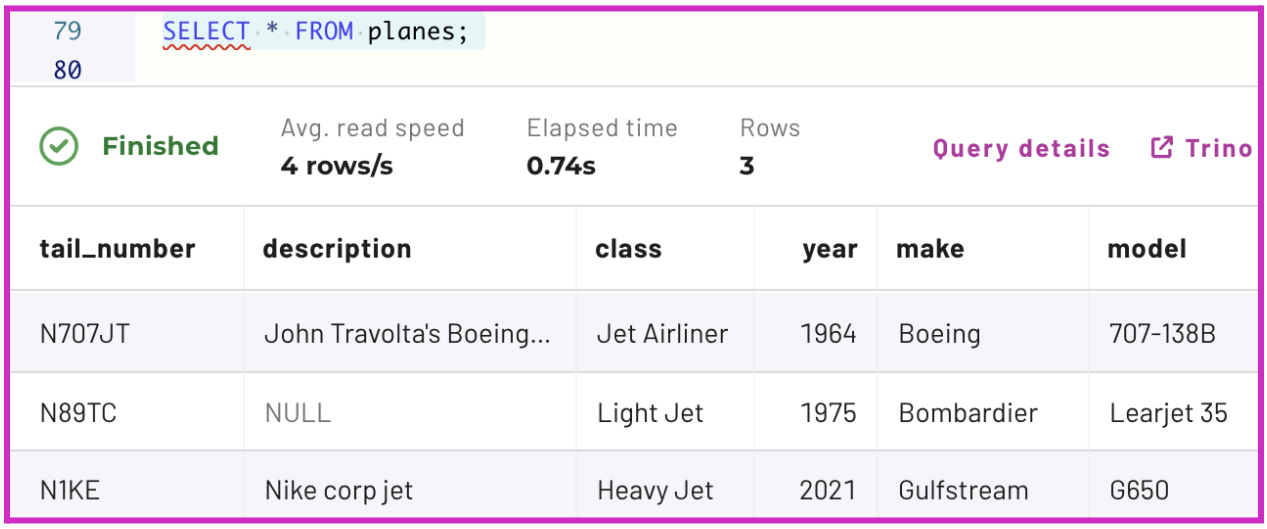
Step 7 - Modify existing records using new column layout

Modify the existing three rows to leverage the new column layout.

```
UPDATE planes
  SET class = 'Jet Airliner',
      year = 1964,
      make = 'Boeing',
      model = '707-138B'
WHERE tail_number = 'N707JT';

UPDATE planes
  SET class = 'Heavy Jet',
      year = 2021,
      make = 'Gulfstream',
      model = 'G650'
WHERE tail_number = 'N1KE';

UPDATE planes
  SET class = 'Light Jet',
      year = 1975,
      make = 'Bombardier',
      model = 'Learjet 35',
      description = null
WHERE tail_number = 'N89TC';
```

When complete, verify that the reporting is correct. The output should resemble the image below.

```
SELECT * FROM planes;
```



tail_number	description	class	year	make	model
N707JT	John Travolta's Boeing...	Jet Airliner	1964	Boeing	707-138B
N89TC	NULL	Light Jet	1975	Bombardier	Learjet 35
N1KE	Nike corp jet	Heavy Jet	2021	Gulfstream	G650

Step 8 - Rename and partition table

Imagine that your company has now decided to carry other types of aircraft, not just planes.

You can use `ALTER TABLE` to rename the `planes` table to allow for additional types, such as helicopters, to be added. The new name for the table will be `aircrafts`.

```
ALTER TABLE planes RENAME TO aircrafts;
```

Also, add partitioning on the `classification` column.

```
ALTER TABLE aircrafts
SET PROPERTIES partitioning = ARRAY['class'];
```

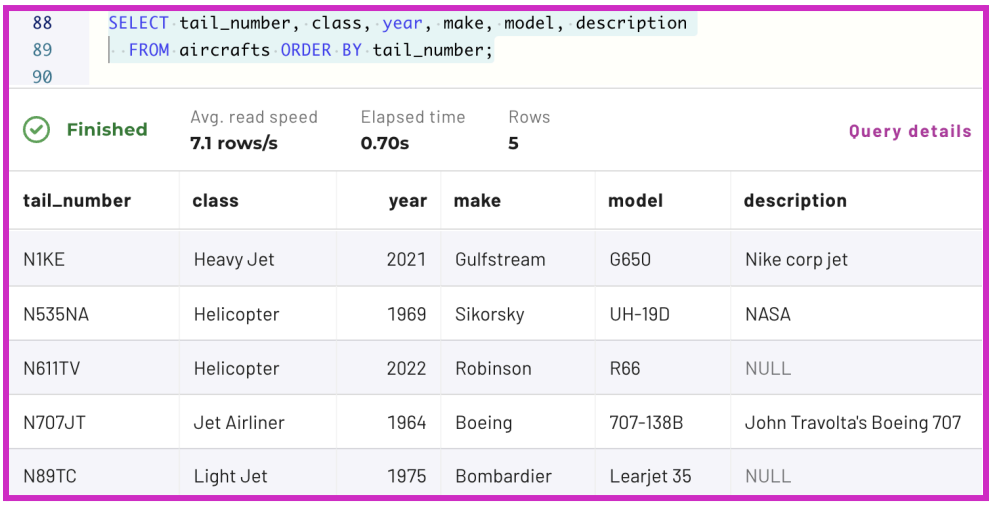
Step 9 - Add two new helicopters into modified table

Add two helicopters to the table as new records using the code below.

```
INSERT INTO aircrafts
 (tail_number, class, year, make, model, description)
VALUES
 ('N535NA', 'Helicopter', 1969, 'Sikorsky', 'UH-19D', 'NASA'),
 ('N611TV', 'Helicopter', 2022, 'Robinson', 'R66', null);
```

When complete, verify that these records have been added using the code below.

```
SELECT tail_number, class, year, make, model, description
  FROM aircrafts ORDER BY tail_number;
```



```
88  SELECT tail_number, class, year, make, model, description
89    FROM aircrafts ORDER BY tail_number;
90
```

Finished | Avg. read speed: 7.1 rows/s | Elapsed time: 0.70s | Rows: 5 | Query details

tail_number	class	year	make	model	description
N1KE	Heavy Jet	2021	Gulfstream	G650	Nike corp jet
N535NA	Helicopter	1969	Sikorsky	UH-19D	NASA
N611TV	Helicopter	2022	Robinson	R66	NULL
N707JT	Jet Airliner	1964	Boeing	707-138B	John Travolta's Boeing 707
N89TC	Light Jet	1975	Bombardier	Learjet 35	NULL

Step 10 - Check partition metadata

Check the `$partitions` metadata table to ensure that the metadata for the new records is being collected properly. There should be two records in the new partition. Use the code below.

```
SELECT partition, record_count, file_count
  FROM "aircrafts$partitions";
```

partition	record_count
{ class = Helicopter }	2

Step 11 - Eliminate partitions

Imagine that the needs of the business change. Fewer aircraft are being purchased than anticipated, and fewer are being added to the table as a result. Upon deeper analysis, you decide that partitioning is no longer necessary because the business need that gave rise to them is no longer true.

Eliminate the partitions using the `ALTER TABLE` command seen below.

```
ALTER TABLE aircrafts
SET PROPERTIES partitioning = ARRAY[];
```

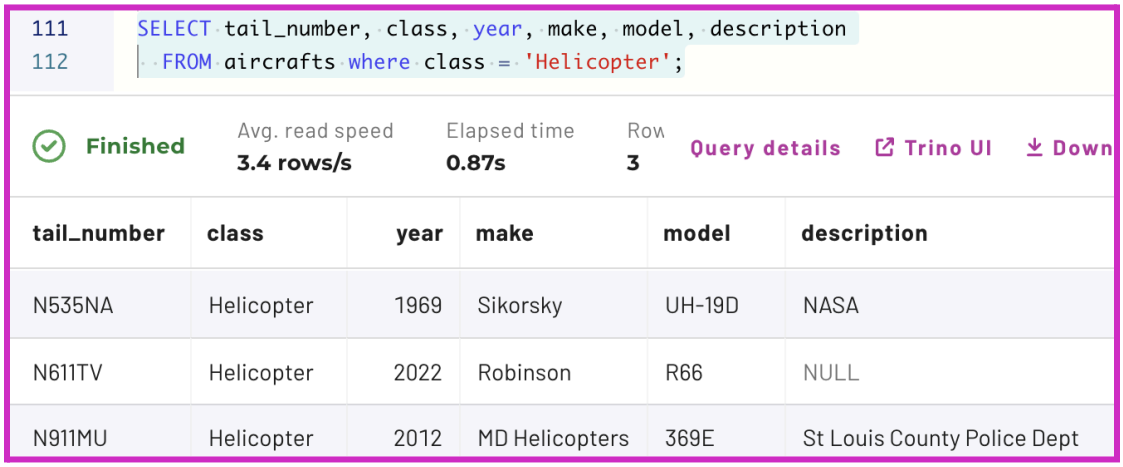
Step 12 - Add new record to table after partitions eliminated

Add another helicopter record.

```
INSERT INTO aircrafts
 (tail_number, class, year, make, model, description)
VALUES
 ('N911MU', 'Helicopter', 2012, 'MD Helicopters', '369E',
  'St Louis County Police Dept');
```

Verify that 3 helicopters are present in the table.

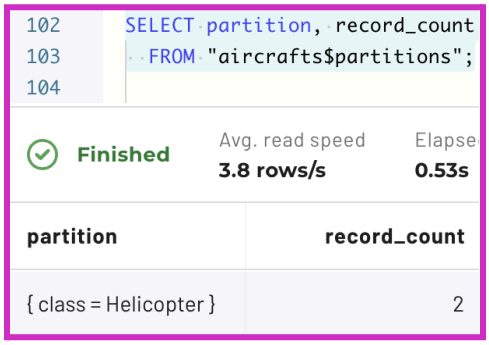
```
SELECT tail_number, class, year, make, model, description
  FROM aircrafts WHERE class = 'Helicopter';
```



tail_number	class	year	make	model	description
N535NA	Helicopter	1969	Sikorsky	UH-19D	NASA
N611TV	Helicopter	2022	Robinson	R66	NULL
N911MU	Helicopter	2012	MD Helicopters	369E	St Louis County Police Dept

Look at the `$partitions` metatable again.

```
SELECT partition, record_count FROM "aircrafts$partitions";
```



partition	record_count
{ class = Helicopter }	2

The new record has been successfully added without a partition, but the original two records remain inside the partition. This is an important point regarding the elimination of partition. Changes to partitions only impact new records but do not automatically alter old records created when the partition was in effect.

Step 13 - Compaction

Take a look at the `$files` metatable.

```
SELECT * FROM "aircrafts$files";
```

content	file_path	file_format	record_count	file_size_in_b...	column_sizes	value_counts	null_value_counts
0	s3://edu-train-galaxy-...	ORC	1	927	NULL	{1=1, 2=1, 4=1, 5=1, 6...	{1=0, 2=1, 4=0, 5=0,..
0	s3://edu-train-galaxy-...	ORC	1	984	NULL	{1=1, 2=1, 4=1, 5=1, 6...	{1=0, 2=0, 4=0, 5=0..
0	s3://edu-train-galaxy-...	ORC	1	1068	NULL	{1=1, 2=1, 4=1, 5=1, 6...	{1=0, 2=0, 4=0, 5=0..
0	s3://edu-train-galaxy-...	ORC	1	1043	NULL	{1=1, 2=1, 4=1, 5=1, 6...	{1=0, 2=0, 4=0, 5=0..
0	s3://edu-train-galaxy-...	ORC	2	693	NULL	{1=2, 2=2, 3=2}	{1=0, 2=0, 3=2}
0	s3://edu-train-galaxy-...	ORC	2	1016	NULL	{1=2, 2=2, 4=2, 5=2...	{1=0, 2=1, 4=0, 5=0,..

The output above shows a large number of small files. These files can be compacted together for better performance.

Run the following command. It will compact any file below a 10MB threshold.

```
ALTER TABLE aircrafts
EXECUTE optimize(file_size_threshold => '10MB');
```

Check the $files metatable again.

```
SELECT * FROM "aircrafts$files";
```

You will see there are fewer files present now using the query below.

```
107  SELECT * FROM "aircrafts$files";
108
```

Finished | Avg. read speed: 3.5 rows/s | Elapsed time: 0.57s | Rows: 2

content	file_path	file_format	record_count	file_size_in_bytes
0	s3://edu-train-galaxy-...	ORC	1	985
0	s3://edu-train-galaxy-...	ORC	5	1319

Note: These files are still very small in size due to the small number of records inserted into them. Nonetheless, the problem has been greatly reduced.

END OF LAB EXERCISE

Cost-based optimizer

Lab 1: The `EXPLAIN` command

Estimated completion time

- 20 minutes

Learning objectives

- In this lesson, you will examine query plans using the SQL command `EXPLAIN`. You will use the `EXPLAIN` command with two different catalogs, hive and MySQL.

Prerequisites

- Student setup - Lab 1: Create student account

Activities

1. Understanding pushdown
2. Using `EXPLAIN` on a relational database
3. Run the query
4. Run the `EXPLAIN` command for the previous query
5. Review `EXPLAIN` plan fragments and operators
6. Review plan Fragment 1
7. Review plan Fragment 0
8. Running `EXPLAIN` on a data lake
9. Run `EXPLAIN` on the query for the first time
10. Review plan Fragment 1

Step 1 - Understanding pushdown

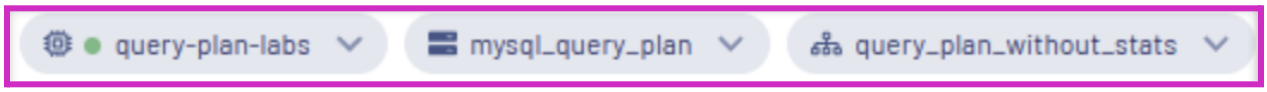
One common term that you will hear when learning about query plans is **pushdown**. Pushdown happens when Starburst pushes a portion of the query processing down to the underlying data source, allowing the data source to complete the processing.

An example of this could be a `WHERE` clause that will filter out some of the rows. It is more efficient to ask the underlying data source to do this, so fewer rows are returned to Starburst. Depending on the connector type, there may not be an underlying process engine. In that case, Starburst has to do the work.

As you work through this lesson, pay attention to any pushdown that is occurring.

Step 2 - Using `EXPLAIN` on a relational database

Log into your Starburst Galaxy account. Expand **Query** and click **Query editor**. Set the cluster to `query-plan-labs`. Set the catalog to `mysql_query_plan`. Set the schema to `query_plan_without_stats`.

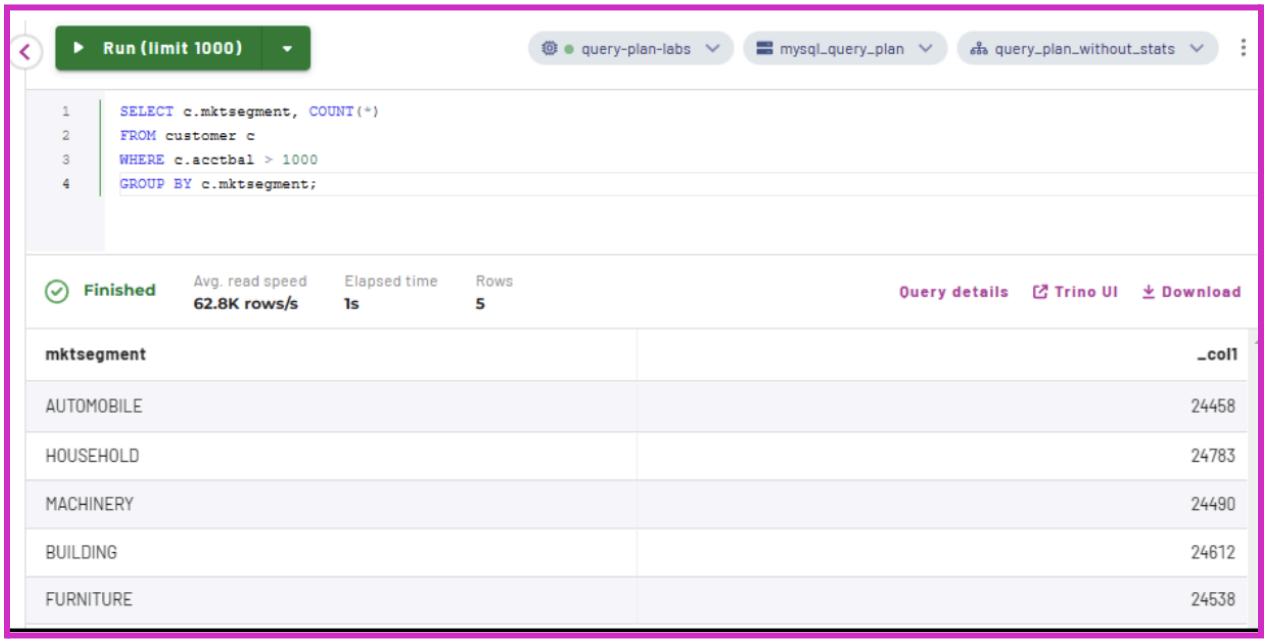


Step 3 - Run the query

This query returns a list, grouped by market segment, of the aggregated count of customers with a balance over $1,000. As you will see in the results, this query returns five total rows, showing that you have five market segments. This is an important thing to remember for later.

Run the following query:

```
SELECT c.mktsegment, COUNT(*)
FROM customer c
WHERE c.acctbal > 1000
GROUP BY c.mktsegment;
```



Step 4 - Run the `EXPLAIN` command for the previous query

Notice that the SQL is almost identical to the previous step; you are just adding EXPLAIN to the beginning.

```
EXPLAIN
SELECT c.mktsegment, COUNT(*)
FROM customer c
WHERE c.acctbal > 1000
GROUP BY c.mktsegment;
```

Step 5 - Review `EXPLAIN` plan fragments and operators

Please note that your exact results may differ from the images shown, depending on the number of nodes you are using.

Begin your analysis with the highest-numbered fragment at the bottom. Always read from the bottom up to review fragments. You should also read from the bottom up within each fragment.

The bottom fragment will always be the `SOURCE` fragment. Fragments that are not `SOURCE` fragments will have a `RemoteSource [X]` Operator, where `X` is the number of the lower fragment that sent (exchanged) data to the current fragment.

```
Fragment 0 [HASH]
    Output layout: [mktsegment, count]
    Output partitioning: SINGLE []
    Output[columnNames = [mktsegment, _col1]]
    │   Layout: [mktsegment:varchar(255), count:bigint]
    │   Estimates: {rows: ? (?), cpu: 0, memory: 0B, network: 0B}
    │   _col1 := count
    └─ Project[]
       │   Layout: [mktsegment:varchar(255), count:bigint]
       │   Estimates: {rows: ? (?), cpu: ?, memory: 0B, network: 0B}
       └─ Aggregate[type = FINAL, keys = [mktsegment], hash = [$hashvalue]]
          │   Layout: [mktsegment:varchar(255), $hashvalue:bigint, \
                 count:bigint]
          │   Estimates: {rows: ? (?), cpu: ?, memory: ?, network: 0B}
          │   count := count("count_0")
          └─ LocalExchange[partitioning = HASH, hashColumn = [$hashvalue], \
                arguments = ["mktsegment"]]
             │   Layout: [mktsegment:varchar(255), count_0:bigint, \
                    $hashvalue:bigint]
             │   Estimates: {rows: ? (?), cpu: ?, memory: 0B, network: 0B}
          2  └─ RemoteSource[sourceFragmentIds = [1]]
                    Layout: [mktsegment:varchar(255), count_0:bigint, \
                          $hashvalue_1:bigint]

Fragment 1 [SOURCE]  1
    Output layout: [mktsegment, count_0, $hashvalue_2]
    Output partitioning: HASH [mktsegment][$hashvalue_2]
```



```
Aggregate[type = PARTIAL, keys = [mktsegment], hash = [$hashvalue_2]]
│   Layout: [mktsegment:varchar(255), $hashvalue_2:bigint, count_0:bigint]
│   count_0 := count(*)
└─ ScanProject[table = mysql_query_plan:query_plan_without_stats. \
        customer query_plan_without_stats.customer constraint on \
              [acctbal] columns=[mktsegment:varchar(255):TINYTEXT]]
       Layout: [mktsegment:varchar(255), $hashvalue_2:bigint]
       Estimates: {rows: ? (?), cpu: ?, memory: 0B, network: 0B}/ \
             {rows: ? (?), cpu: ?, memory: 0B, network: 0B}
       $hashvalue_2 := combine_hash(bigint '0', COALESCE( \
             "$operator$hash_code"("mktsegment"), 0))
       mktsegment := mktsegment:varchar(255):TINYTEXT
```

1 SOURCE fragment

The bottom fragment will always be the `SOURCE` fragment. It is also the highest-numbered fragment. Query plans should be read from the bottom up.

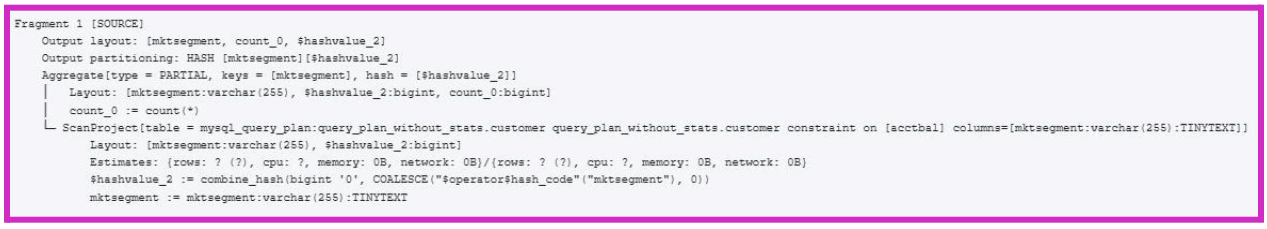
2 Remote source operator

Fragments that are not `SOURCE` fragments will have a `RemoteSource [X]` operator, where `X` is the number of the lower fragment exchanged (sent) data to the current fragment. So, in this case, `Fragment 1` sent data to `Fragment 0`.

Step 6 - Review plan Fragment 1

Review the labeled graphic of Fragment 1 below. Click on each plus sign to learn more about the corresponding section of the fragment.

Note: Don't forget to read from the bottom up.



Click here to explore the query plan fragment.

Step 7 - Review plan Fragment 0

Review the labeled graphic of Fragment 0 below. Click on each plus sign to learn more about the corresponding section of the fragment.

Note: Don't forget to read from the bottom up.

```
Fragment 0 [HASH]
    Output layout: [mktsegment, count]
    Output partitioning: SINGLE []
    Output[columnNames = [mktsegment, _col1]]
    │   Layout: [mktsegment:varchar(255), count:bigint]
    │   Estimates: {rows: ? (?), cpu: 0, memory: 0B, network: 0B}
    │   _col1 := count
    └─ Project[]
       │   Layout: [mktsegment:varchar(255), count:bigint]
       │   Estimates: {rows: ? (?), cpu: ?, memory: 0B, network: 0B}
       └─ Aggregate[type = FINAL, keys = [mktsegment], hash = [$hashvalue]]
          │   Layout: [mktsegment:varchar(255), $hashvalue:bigint, count:bigint]
          │   Estimates: {rows: ? (?), cpu: ?, memory: ?, network: 0B}
          │   count := count("count_0")
          └─ LocalExchange[partitioning = HASH, hashColumn = [$hashvalue], arguments = ["mktsegment"]]
             │   Layout: [mktsegment:varchar(255), count_0:bigint, $hashvalue:bigint]
             │   Estimates: {rows: ? (?), cpu: ?, memory: 0B, network: 0B}
             └─ RemoteSource[sourceFragmentIds = [1]]
                    Layout: [mktsegment:varchar(255), count_0:bigint, $hashvalue_1:bigint]
```

Click here to explore the query plan fragment.

Step 8 - Running `EXPLAIN` on a data lake

In this section, you'll run `EXPLAIN` on the same query that you ran in the last section. This time, however, you will be querying a data lake rather than a relational database. The explanations provided will focus on the differences between hive and what you saw with MySQL.

Open another browser tab and log in to your Starburst Galaxy account. Expand **Query** and click **Query editor**.

Now set the `cluster to query-plan-labs`. Set the catalog to `lakehouse_query_plan`. Set the schema to `query_plan_without_stats`.

Step 9 - Run `EXPLAIN` on the query for the first time

Use `EXPLAIN` on the query using the code below.

```
EXPLAIN
SELECT c.mktsegment, COUNT(*)
FROM customer c
WHERE c.acctbal > 1000
GROUP BY c.mktsegment;
```

Step 10 - Review plan Fragment 1

Fragment 0 is identical to what you saw above with the relational database. Fragment 1 has a slightly different operator, as you'll see now.

Review the labeled graphic of Fragment 1 below. Click on each plus sign to learn more about the corresponding section of the fragment.

Note: Don't forget to read from the bottom up.

Click here to explore the query plan fragment.

The `EXPLAIN` command is a convenient tool used to view a logical query plan without actually executing the query. Query plans are composed of numbered fragments. When reviewing a query plan, you must always start from the bottom, or `SOURCE`, fragment.

Knowing how to interpret this output is critical to performance tuning. In the next lesson, you will get some more practice with query plans, this time using the `EXPLAIN ANALYZE` command.

END OF LAB EXERCISE

Lab 2: The `EXPLAIN ANALYZE` command

Estimated completion time

- 20 minutes

Learning objectives

- In this lesson, you will build upon your knowledge of query plans by examining the output of the `SQL EXPLAIN ANALYZE` command. You will once again compare the query plan output for a relational database and a data lake.

Prerequisites

- Student setup - Lab 1: Create student account

Activities

1. Understanding `EXPLAIN ANALYZE`
2. Set up relational database environment
3. Run the `EXPLAIN ANALYZE` command for a query in MySQL
4. Review plan Fragment 2
5. Review plan Fragment 1
6. Run `EXPLAIN ANALYZE` on a data lake
7. Set up your data lake environment
8. Run `EXPLAIN ANALYZE` for the query from the first section
9. Review plan Fragment 2

Step 1 - Understanding `EXPLAIN ANALYZE`

`EXPLAIN ANALYZE` differs from `EXPLAIN` in that it not only provides the query plan but executes the query as well. You will see some additional information in the query plan output.

It includes the extra execution metrics that illustrate what occurred when the SQL ran.

It does not include a Fragment 0 because the result of an `EXPLAIN ANALYZE` is the plan itself. This is because the system does not output the results of the SQL, which nullifies the need for a Fragment 0.

Step 2 - Set up relational database environment

Begin by setting up your environment using the following steps:

- Sign in to your Starburst Galaxy account.
- Expand **Query** and click **Query editor**.
- Set the cluster to **query-plan-labs**.
- Set the catalog to **mysql_query_plan**.
- Set the schema to **query_plan_without_stats**.

Step 3 - Run the `EXPLAIN ANALYZE` command for a query in MySQL

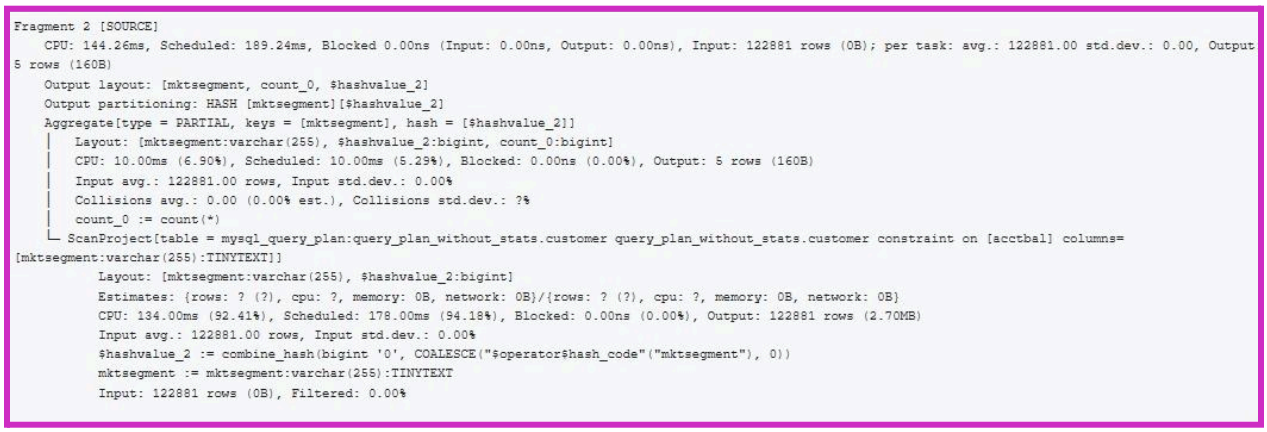
Notice that the query is the same one you used in the last lesson.

```
EXPLAIN ANALYZE
SELECT c.mktsegment, COUNT(*)
FROM customer c
WHERE c.acctbal > 1000
GROUP BY c.mktsegment;
```

Step 4 - Review plan Fragment 2

Review the labeled graphic of Fragment 2 below. Click on each plus sign to learn more about the corresponding section of the fragment.

Note: Don't forget to read from the bottom up.



Click here to explore the query plan fragment.

Step 5 - Review plan Fragment 1

Notice that the final aggregation of the hashes is a very small part of the overall processing time.

You should see the same additional information as in Fragment 2.

```
Fragment 1 [HASH]
    CPU: 3.17ms, Scheduled: 3.23ms, Blocked 769.57ms (Input: 385.00ms, Output: 0.00ns), Input: 5 rows (160B); per task: avg.: 5.00 std.dev.: 0.00, Output: 5 rows
(115B)
    Output layout: [mktsegment, count]
    Output partitioning: SINGLE []
    Project[]
    │   Layout: [mktsegment:varchar(255), count:bigint]
    │   Estimates: {rows: ? (?), cpu: ?, memory: 0B, network: 0B}
    │   CPU: 0.00ns (0.00%), Scheduled: 0.00ns (0.00%), Blocked: 0.00ns (0.00%), Output: 5 rows (115B)
    │   Input avg.: 2.50 rows, Input std.dev.: 60.00%
    └─ Aggregate[type = FINAL, keys = [mktsegment], hash = [$hashvalue]]
       │   Layout: [mktsegment:varchar(255), $hashvalue:bigint, count:bigint]
       │   Estimates: {rows: ? (?), cpu: ?, memory: ?, network: 0B}
       │   CPU: 1.00ms (0.69%), Scheduled: 1.00ms (0.53%), Blocked: 0.00ns (0.00%), Output: 5 rows (160B)
       │   Input avg.: 2.50 rows, Input std.dev.: 60.00%
       │   Collisions avg.: 0.00 (0.00% est.), Collisions std.dev.: ?%
       │   count := count("count_0")
       └─ LocalExchange[partitioning = HASH, hashColumn = [$hashvalue], arguments = ["mktsegment"]]
          │   Layout: [mktsegment:varchar(255), count_0:bigint, $hashvalue:bigint]
          │   Estimates: {rows: ? (?), cpu: ?, memory: 0B, network: 0B}
          │   CPU: 0.00ns (0.00%), Scheduled: 0.00ns (0.00%), Blocked: 385.00ms (50.00%), Output: 5 rows (160B)
          │   Input avg.: 2.50 rows, Input std.dev.: 100.00%
          └─ RemoteSource[sourceFragmentIds = [2]]
                Layout: [mktsegment:varchar(255), count_0:bigint, $hashvalue_1:bigint]
                CPU: 0.00ns (0.00%), Scheduled: 0.00ns (0.00%), Blocked: 385.00ms (50.00%), Output: 5 rows (160B)
                Input avg.: 2.50 rows, Input std.dev.: 100.00%
```

Step 6 - Run `EXPLAIN ANALYZE` on a data lake

In this section, you'll run `EXPLAIN ANALYZE` on the same query that you ran in the last section. This time, however, you will be querying a data lake rather than a relational database. The explanations provided will focus on the differences between hive and what you saw with MySQL.

Step 7 - Set up your data lake environment

Begin by setting up your environment using the following steps:

- Open another browser tab and log in to your Starburst Galaxy account
- Expand **Query** and click **Query editor**.
- Set the cluster to **query-plan-labs**.
- Set the catalog to **lakehouse_query_plan**.
- Set the schema to **query_plan_without_stats.**

Step 8 - Run `EXPLAIN ANALYZE` for the query from the first section

Run the following SQL:

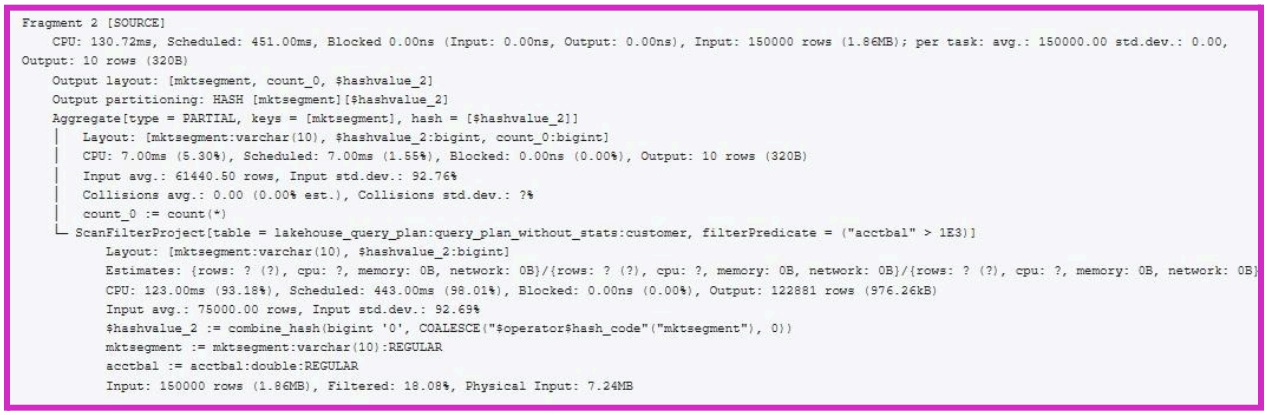
```
EXPLAIN ANALYZE
SELECT c.mktsegment, COUNT(*)
FROM customer c
WHERE c.acctbal > 1000
GROUP BY c.mktsegment;
```


Step 9 - Review plan Fragment 2

There are several interesting differences between the `EXPLAIN ANALYZE` running on MySQL and Hive.

Review the labeled graphic of Fragment 2 below. Click on each plus sign to learn more about the corresponding section of the fragment.

Note: Don't forget to read from the bottom up.



Click here to explore the query plan fragment.

END OF LAB EXERCISE

Lab 3: Explore the impact of statistics on query plans

Estimated completion time

- 45 minutes

Learning objectives

- In this lesson, you will see how the Starburst Cost-based optimizer (CBO) works with `SQL JOIN` functions. You'll write `JOIN` functions for tables with and without statistics and explore their functionality with the `EXPLAIN` and `EXPLAIN ANALYZE` commands. How you join datasets can have a huge impact on query performance, so optimizing queries that join data sets is a critical part of using Starburst. You will once again compare the query plan output for a relational database and a data lake.

Prerequisites

- Student setup - Lab 1: Create student account

Activities

1. Set up data lake environment without stats
2. Run the query that joins two tables
3. Run `EXPLAIN ANALYZE` for the query
4. Review plan Fragment 4
5. Review plan Fragment 3
6. Review plan Fragment 2
7. Review plan Fragment 1
8. Set up environment with stats
9. View the table stats
10. Run `EXPLAIN ANALYZE` with stats
11. Review plan Fragment 3
12. Review plan Fragment 2
13. Review plan Fragment 1
14. Run `EXPLAIN ANALYZE` on a query with a function

Step 1 - Set up environment without stats

Begin by setting up your environment using the following steps:

- Log into your Starburst Galaxy account.
- Expand **Query** and click **Query editor**.
- Set the cluster to **query-plan-labs**.
- Set the catalog to **lakehouse_query_plan**.
- Set the schema to **query_plan_without_stats**.

Step 2 - Run the query that joins two tables

Start by running the query that you will be using in this section. The purpose of this query is to return a list of the aggregated count of orders where the customers' account balance is over $5,000. The result will be grouped by order status and should show three rows.

```
SELECT o.orderstatus, COUNT(*)
FROM customer c JOIN orders o ON (c.custkey = o.custkey)
WHERE c.acctbal > 5000
GROUP BY o.orderstatus;
```

Step 3 - Run `EXPLAIN ANALYZE` for the query

Run the following SQL to generate the query plan.

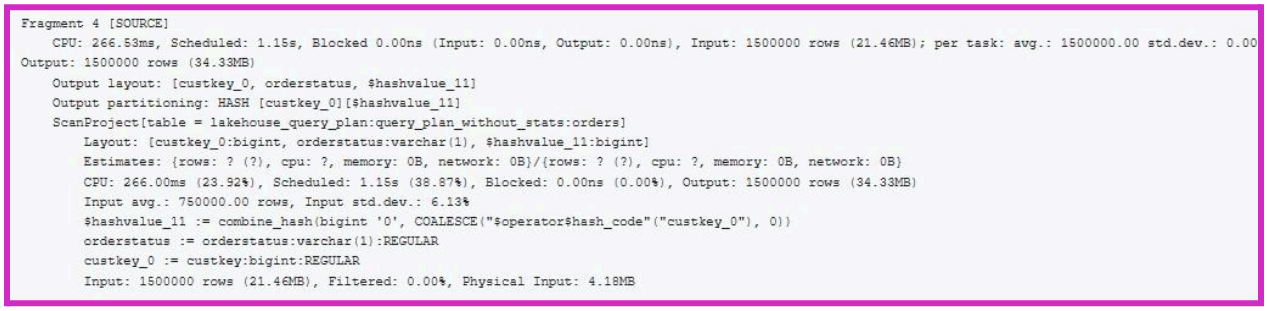
```
EXPLAIN ANALYZE
SELECT o.orderstatus, COUNT(*)
FROM customer c JOIN orders o ON (c.custkey = o.custkey)
WHERE c.acctbal > 5000
GROUP BY o.orderstatus;
```

Step 4 - Review plan Fragment 4

Fragment 4 is one of two `SOURCE` fragments for this query plan. There are two because the query includes two tables. Fragment 4 is gathering data from the `orders` table, while Fragment 3, as you'll see, is gathering data from the `customer` table.

Review the labeled graphic of Fragment 4 below. Click on each plus sign to learn more about the corresponding section of the fragment.

Note: Don't forget to read from the bottom up.



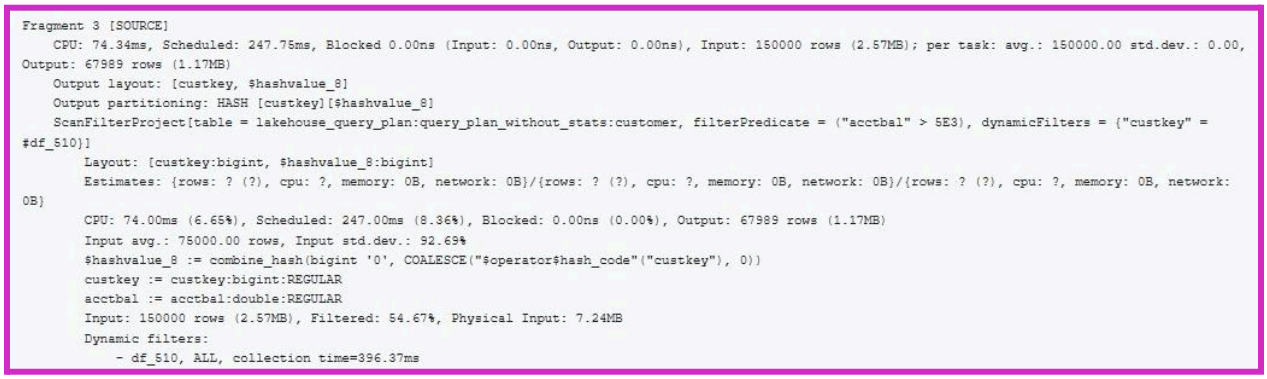
Click here to explore the query plan fragment.

Step 5 - Review plan Fragment 3

Again, Fragment 3 is responsible for the customer table.

Review the labeled graphic of Fragment 3 below. Click on each plus sign to learn more about the corresponding section of the fragment.

Note: Don't forget to read from the bottom up.



```
Fragment 3 [SOURCE]
    CPU: 74.34ms, Scheduled: 247.75ms, Blocked 0.00ns (Input: 0.00ns, Output: 0.00ns), Input: 150000 rows (2.57MB); per task: avg.: 150000.00 std.dev.: 0.00,
Output: 67989 rows (1.17MB)
    Output layout: [custkey, $hashvalue_8]
    Output partitioning: HASH [custkey][$hashvalue_8]
    ScanFilterProject[table = lakehouse_query_plan:query_plan_without_stats:customer, filterPredicate = ("acctbal" > 5E3), dynamicFilters = {"custkey" =
#df_510}]
        Layout: [custkey:bigint, $hashvalue_8:bigint]
        Estimates: {rows: ? (?), cpu: ?, memory: 0B, network: 0B}/{rows: ? (?), cpu: ?, memory: 0B, network: 0B}/{rows: ? (?), cpu: ?, memory: 0B, network:
0B}
        CPU: 74.00ms (6.65%), Scheduled: 247.00ms (8.36%), Blocked: 0.00ns (0.00%), Output: 67989 rows (1.17MB)
        Input avg.: 75000.00 rows, Input std.dev.: 92.69%
        $hashvalue_8 := combine_hash(bigint '0', COALESCE("$operator$hash_code"("custkey"), 0))
        custkey := custkey:bigint:REGULAR
        acctbal := acctbal:double:REGULAR
        Input: 150000 rows (2.57MB), Filtered: 54.67%, Physical Input: 7.24MB
        Dynamic filters:
            - df_510, ALL, collection time=396.37ms
```

Click here to explore the query plan fragment.

Step 6 - Review plan Fragment 2

Review the labeled graphic of Fragment 2 below. Click on each plus sign to learn more about the corresponding section of the fragment.

Note: Don't forget to read from the bottom up.

```
Fragment 2 [HASH]
    CPU: 774.29ms, Scheduled: 1.56s, Blocked 4.01s (Input: 1.21s, Output: 0.00ns), Input: 1567989 rows (35.50MB); per task: avg.: 1567989.00 std.dev.: 0.00,
Output: 6 rows (144B)
    Output layout: [orderstatus, count_5, $hashvalue_12]
    Output partitioning: HASH [orderstatus][$hashvalue_12]
    Aggregate[type = PARTIAL, keys = [orderstatus], hash = [$hashvalue_12]]
    │   Layout: [orderstatus:varchar(1), $hashvalue_12:bigint, count_5:bigint]
    │   CPU: 43.00ms (3.87%), Scheduled: 43.00ms (1.45%), Blocked: 0.00ns (0.00%), Output: 6 rows (144B)
    │   Input avg.: 339774.00 rows, Input std.dev.: 6.29%
    │   Collisions avg.: 0.00 (0.00% est.), Collisions std.dev.: ?%
    │   count_5 := count(*)
    └─ Project[]
       │   Layout: [orderstatus:varchar(1), $hashvalue_12:bigint]
       │   Estimates: {rows: ? (?), cpu: ?, memory: 0B, network: 0B}
       │   CPU: 17.00ms (1.53%), Scheduled: 57.00ms (1.93%), Blocked: 0.00ns (0.00%), Output: 679548 rows (9.72MB)
       │   Input avg.: 339774.00 rows, Input std.dev.: 6.29%
       │   $hashvalue_12 := combine_hash(bigint '0', COALESCE("$operator$hash_code"("orderstatus"), 0))
       └─ InnerJoin[criteria = ("custkey" = "custkey_0"), hash = [$hashvalue_7, $hashvalue_9], distribution = PARTITIONED]
          │   Layout: [orderstatus:varchar(1)]
          │   Estimates: {rows: ? (?), cpu: ?, memory: ?, network: 0B}
          │   CPU: 532.00ms (47.84%), Scheduled: 850.00ms (28.76%), Blocked: 1.85s (20.97%), Output: 679548 rows (3.89MB)
          │   Left (probe) Input avg.: 33994.50 rows, Input std.dev.: 5.91%
          │   Right (build) Input avg.: 750000.00 rows, Input std.dev.: 0.00%
          │   Collisions avg.: 8446.98 (4.05% est.), Collisions std.dev.: 5.75%
          │   Distribution: PARTITIONED
          │   dynamicFilterAssignments = {custkey_0 -> #df_510}
          ├─ RemoteSource[sourceFragmentIds = [3]]
          │      Layout: [custkey:bigint, $hashvalue_7:bigint]
          │      CPU: 0.00ns (0.00%), Scheduled: 0.00ns (0.00%), Blocked: 399.00ms (4.53%), Output: 67989 rows (1.17MB)
          │      Input avg.: 33994.50 rows, Input std.dev.: 5.91%
          └─ LocalExchange[partitioning = HASH, hashColumn = [$hashvalue_9], arguments = ["custkey_0"]]
             │   Layout: [custkey_0:bigint, orderstatus:varchar(1), $hashvalue_9:bigint]
             │   Estimates: {rows: ? (?), cpu: ?, memory: 0B, network: 0B}
             │   CPU: 171.00ms (15.38%), Scheduled: 549.00ms (18.57%), Blocked: 1.00s (11.34%), Output: 1500000 rows (34.33MB)
             │   Input avg.: 750000.00 rows, Input std.dev.: 2.95%
             └─ RemoteSource[sourceFragmentIds = [4]]
                    Layout: [custkey_0:bigint, orderstatus:varchar(1), $hashvalue_10:bigint]
                    CPU: 8.00ms (0.72%), Scheduled: 60.00ms (2.03%), Blocked: 808.00ms (9.17%), Output: 1500000 rows (34.33MB)
                    Input avg.: 750000.00 rows, Input std.dev.: 2.95%
```

Click here to explore the query plan fragment.

Step 7 - Review plan Fragment 1

Review the labeled graphic below, which only has one item called out.

```
Fragment 1 [HASH]
    CPU: 2.92ms, Scheduled: 2.99ms, Blocked 4.75s (Input: 2.38s, Output: 0.00ns), Input: 6 rows (144B); per task: avg.: 6.00 std.dev.: 0.00, Output: 3 rows
(45B)
    Output layout: [orderstatus, count]
    Output partitioning: SINGLE []
    Project[]
    │   Layout: [orderstatus:varchar(1), count:bigint]
    │   Estimates: {rows: ? (?), cpu: ?, memory: 0B, network: 0B}
    │   CPU: 0.00ns (0.00%), Scheduled: 0.00ns (0.00%), Blocked: 0.00ns (0.00%), Output: 3 rows (45B)
    │   Input avg.: 1.50 rows, Input std.dev.: 33.33%
    └─ Aggregate[type = FINAL, keys = [orderstatus], hash = [$hashvalue]]
       │   Layout: [orderstatus:varchar(1), $hashvalue:bigint, count:bigint]
       │   Estimates: {rows: ? (?), cpu: ?, memory: ?, network: 0B}
       │   CPU: 1.00ms (0.09%), Scheduled: 1.00ms (0.03%), Blocked: 0.00ns (0.00%), Output: 3 rows (72B)
       │   Input avg.: 3.00 rows, Input std.dev.: 33.33%
       │   Collisions avg.: 0.00 (0.00% est.), Collisions std.dev.: ?%
       │   count := count("count_5")
       └─ LocalExchange[partitioning = HASH, hashColumn = [$hashvalue], arguments = ["orderstatus"]]
          │   Layout: [orderstatus:varchar(1), count_5:bigint, $hashvalue:bigint]
          │   Estimates: {rows: ? (?), cpu: ?, memory: 0B, network: 0B}
          │   CPU: 0.00ns (0.00%), Scheduled: 0.00ns (0.00%), Blocked: 2.38s (27.00%), Output: 6 rows (144B)
          │   Input avg.: 3.00 rows, Input std.dev.: 0.00%
          └─ RemoteSource[sourceFragmentIds = [2]]
                Layout: [orderstatus:varchar(1), count_5:bigint, $hashvalue_6:bigint]
                CPU: 0.00ns (0.00%), Scheduled: 0.00ns (0.00%), Blocked: 2.38s (27.00%), Output: 6 rows (144B)
                Input avg.: 3.00 rows, Input std.dev.: 0.00%
```

Click here to explore the query plan fragment.

Note: Because the tables have no statistics, the Query Plan you see is not as efficient as it should be.

- Without statistics for ALL tables, our CBO becomes an RBO (a rule-based optimizer) which runs the joins in the order written.
- If there were table stats (number of rows), the CBO would still decide between a Distributed/Partitioned vs. Broadcast/Replicated join strategy for each pair of tables being joined, provided that one of them is small enough to be considered for broadcasting.
- With the number of rows stats, the CBO would also decide on which side a given table should be, i.e., left/right (probe/build or outer/inner).

Step 8 - Set up environment with stats

Now that you've seen some limitations that are present when tables don't have statistics, let's see what happens when tables do have statistics. Statistics for Hive catalogs are stored in the metastore. Starburst Galaxy stores this information in the same manner as Hive. This means that any statistics can be read as long as they are in hive format.

Begin by setting up your environment using the following steps:

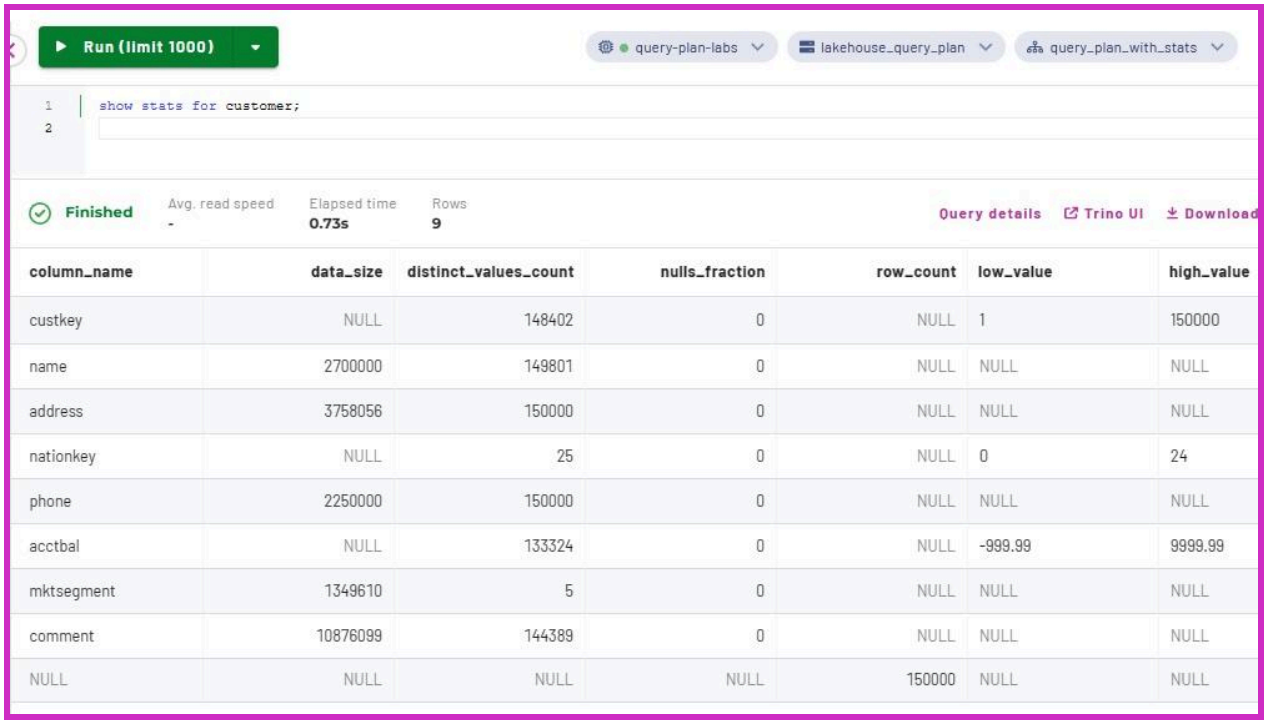
- Log into your Starburst Galaxy account.
- Expand **Query** and click **Query editor**.
- Set the cluster to **query-plan-labs**.
- Set the catalog to **lakehouse_query_plan**.
- Set the schema to **query_plan_with_stats**. Notice this time, it is a different schema with stats.

Step 9 - View the table stats

Run the following `SHOW STATS` commands, one at a time, to view the stats from the customer and orders tables.

```
SHOW STATS FOR customer;
SHOW STATS FOR orders;
```

Review the labeled graphic below, which shows the stats for the `customer` table. Click each plus sign to learn more about the corresponding item.



column_name	data_size	distinct_values_count	nulls_fraction	row_count	low_value	high_value
custkey	NULL	148402	0	NULL	1	150000
name	2700000	149801	0	NULL	NULL	NULL
address	3758056	150000	0	NULL	NULL	NULL
nationkey	NULL	25	0	NULL	0	24
phone	2250000	150000	0	NULL	NULL	NULL
acctbal	NULL	133324	0	NULL	-999.99	9999.99
mktsegment	1349610	5	0	NULL	NULL	NULL
comment	10876099	144389	0	NULL	NULL	NULL
NULL	NULL	NULL	NULL	150000	NULL	NULL

Click here to explore the table stats.

Step 10 - Run EXPLAIN ANALYZE with stats

Now that you've collected and observed the stats, you can run the `EXPLAIN ANALYZE` command below. Notice that you now have one less fragment than when you ran the same query on hive without statistics.

```
EXPLAIN ANALYZE
SELECT o.orderstatus, COUNT(*)
FROM customer c JOIN orders o ON (c.custkey = o.custkey)
WHERE c.acctbal > 5000
GROUP BY o.orderstatus;
```

Step 11 - Review plan Fragment 3

Review the labeled graphic of Fragment 3 below. Click on each plus sign to learn more about the corresponding section of the fragment.

Note: Don't forget to read from the bottom up.

```
Fragment 3 [SOURCE]
    CPU: 51.56ms, Scheduled: 380.21ms, Blocked 0.00ns (Input: 0.00ns, Output: 0.00ns), Input: 150000 rows (2.57MB); per task: avg.: 150000.00 std.dev.: 0.00, Output: 67989
rows (1.17MB)
    Output layout: [custkey, $hashvalue_10]
    Output partitioning: BROADCAST []
    ScanFilterProject[table = lakehouse_query_plan:query_plan_with_stats:customer, filterPredicate = ("acctbal" > 5E3)]
        Layout: [custkey:bigint, $hashvalue_10:bigint]
        Estimates: {rows: 150000 (2.57MB), cpu: 2.57M, memory: 0B, network: 0B}/{rows: 68182 (1.17MB), cpu: 2.57M, memory: 0B, network: 0B}/{rows: 68182 (1.17MB), cpu:
1.17M, memory: 0B, network: 0B}
        CPU: 52.00ms (7.93%), Scheduled: 380.00ms (17.34%), Blocked: 0.00ns (0.00%), Output: 67989 rows (1.17MB)
        Input avg.: 75000.00 rows, Input std.dev.: 92.69%
        $hashvalue_10 := combine_hash(bigint '0', COALESCE("$operator$hash_code"("custkey"), 0))
        custkey := custkey:bigint:REGULAR
        acctbal := acctbal:double:REGULAR
        Input: 150000 rows (2.57MB), Filtered: 54.67%, Physical Input: 7.24MB
```

[Click here to explore the query plan fragment]

Step 12 - Review plan Fragment 2

Review the labeled graphic of Fragment 2 below. Click on each plus sign to learn more about the corresponding section of the fragment.

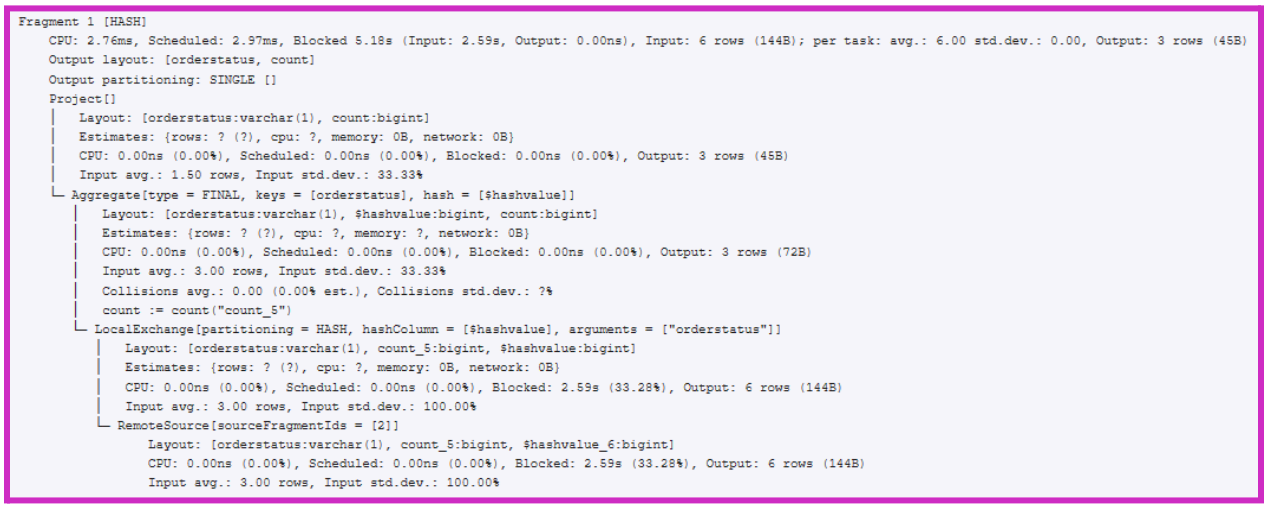
Note: Don't forget to read from the bottom up.

```
Fragment 2 [SOURCE]
    CPU: 607.32ms, Scheduled: 1.82s, Blocked 2.60s (Input: 721.36ms, Output: 0.00ns), Input: 1567989 rows (22.62MB); per task: avg.: 1567989.00 std.dev.: 0.00, Output: 6
rows (144B)
    Output layout: [orderstatus, count_5, $hashvalue_11]
    Output partitioning: HASH [orderstatus][$hashvalue_11]
    Aggregate[type = PARTIAL, keys = [orderstatus], hash = [$hashvalue_11]]
    │   Layout: [orderstatus:varchar(1), $hashvalue_11:bigint, count_5:bigint]
    │   CPU: 88.00ms (13.41%), Scheduled: 88.00ms (4.01%), Blocked: 0.00ns (0.00%), Output: 6 rows (144B)
    │   Input avg.: 339774.00 rows, Input std.dev.: 6.10%
    │   Collisions avg.: 0.00 (0.00% est.), Collisions std.dev.: ?%
    │   count_5 := count(*)
    └─ Project[]
       │   Layout: [orderstatus:varchar(1), $hashvalue_11:bigint]
       │   Estimates: {rows: 1006077 (14.39MB), cpu: 14.39M, memory: 0B, network: 0B}
       │   CPU: 116.00ms (17.68%), Scheduled: 151.00ms (6.89%), Blocked: 0.00ns (0.00%), Output: 679548 rows (12.31MB)
       │   Input avg.: 339774.00 rows, Input std.dev.: 6.10%
       │   $hashvalue_11 := combine_hash(bigint '0', COALESCE("$operator$hash_code"("orderstatus"), 0))
       └─ InnerJoin[criteria = ("custkey_0" = "custkey"), hash = [$hashvalue_7, $hashvalue_8], distribution = REPLICATED]
          │   Layout: [orderstatus:varchar(1)]
          │   Estimates: {rows: 1006077 (5.76MB), cpu: 41.26M, memory: 1.17MB, network: 0B}
          │   CPU: 231.00ms (35.21%), Scheduled: 321.00ms (14.64%), Blocked: 1.52s (19.56%), Output: 679548 rows (6.48MB)
          │   Left (probe) Input avg.: 750000.00 rows, Input std.dev.: 6.13%
          │   Right (build) Input avg.: 67989.00 rows, Input std.dev.: 0.00%
          │   Collisions avg.: 5032.00 (99.29% est.), Collisions std.dev.: 0.00%
          │   Distribution: REPLICATED
          │   dynamicFilterAssignments = {custkey -> #df_506}
          ├─ ScanFilterProject[table = lakehouse_query_plan:query_plan_with_stats:orders, dynamicFilters = {"custkey_0" = #df_506}]
          │      Layout: [custkey_0:bigint, orderstatus:varchar(1), $hashvalue_7:bigint]
          │      Estimates: {rows: 1500000 (34.33MB), cpu: 21.46M, memory: 0B, network: 0B}/{rows: 1500000 (34.33MB), cpu: 21.46M, memory: 0B, network: 0B}/{rows: 1500000
(34.33MB), cpu: 34.33M, memory: 0B, network: 0B}
          │      CPU: 167.00ms (25.46%), Scheduled: 1.25s (57.03%), Blocked: 0.00ns (0.00%), Output: 1500000 rows (34.33MB)
          │      Input avg.: 750000.00 rows, Input std.dev.: 6.13%
          │      $hashvalue_7 := combine_hash(bigint '0', COALESCE("$operator$hash_code"("custkey_0"), 0))
          │      orderstatus := orderstatus:varchar(1):REGULAR
          │      custkey_0 := custkey:bigint:REGULAR
          │      Input: 1500000 rows (21.46MB), Filtered: 0.00%, Physical Input: 4.18MB
          │      Dynamic filters:
          │          - df_506, ALL, collection time=189.42ms
          └─ LocalExchange[partitioning = SINGLE]
             │   Layout: [custkey:bigint, $hashvalue_8:bigint]
             │   Estimates: {rows: 68182 (1.17MB), cpu: 0, memory: 0B, network: 0B}
             │   CPU: 1.00ms (0.15%), Scheduled: 1.00ms (0.05%), Blocked: 359.00ms (4.61%), Output: 67989 rows (1.17MB)
             │   Input avg.: 33994.50 rows, Input std.dev.: 4.71%
             └─ RemoteSource[sourceFragmentIds = [3]]
                   Layout: [custkey:bigint, $hashvalue_9:bigint]
                   CPU: 1.00ms (0.15%), Scheduled: 1.00ms (0.05%), Blocked: 721.00ms (9.26%), Output: 67989 rows (1.17MB)
                   Input avg.: 33994.50 rows, Input std.dev.: 4.71%
```

Click here to explore the query plan fragment.

Step 13 - Review plan Fragment 1

This fragment only has one item called out. Review the labeled graphic below for more information.



Click here to explore the query plan fragment.

Step 14 - Run `EXPLAIN ANALYZE` on a query with a function

Now it's time to observe what happens when a function is added to a query. At the time of this writing, Starburst cannot determine the impact of functions, which you'll see after reviewing the query plan generated from the following query. Notice the addition of the `ROUND` function:

```
EXPLAIN ANALYZE
SELECT o.orderstatus, COUNT(*)
FROM customer c JOIN orders o ON (c.custkey = o.custkey)
WHERE round(c.acctbal) > 5000
GROUP BY o.orderstatus;
```

Step 15 - Review plan Fragment 3

Review the labeled graphic of Fragment 3 below. Notice that it looks very similar to the one without stats:


```
Fragment 3 [SOURCE]
    CPU: 190.50ms, Scheduled: 1.33s, Blocked 0.00ns (Input: 0.00ns, Output: 0.00ns), Input: 1500000 rows (21.46MB); per task: avg.: 1500000.00 std.dev.: 0.00, Output:
1500000 rows (34.33MB)
    Output layout: [custkey_0, orderstatus, $hashvalue_10]
    Output partitioning: BROADCAST []
    ScanProject[table = lakehouse_query_plan:query_plan_with_stats:orders]
        Layout: [custkey_0:bigint, orderstatus:varchar(1), $hashvalue_10:bigint]
        Estimates: {rows: 1500000 (34.33MB), cpu: 21.46M, memory: 0B, network: 0B}/{rows: 1500000 (34.33MB), cpu: 34.33M, memory: 0B, network: 0B}
        CPU: 191.00ms (20.28%), Scheduled: 1.33s (48.67%), Blocked: 0.00ns (0.00%), Output: 1500000 rows (34.33MB)
        Input avg.: 750000.00 rows, Input std.dev.: 6.13%
        $hashvalue_10 := combine_hash(bigint '0', COALESCE("$operator$hash_code"("custkey_0"), 0))
        orderstatus := orderstatus:varchar(1):REGULAR
        custkey_0 := custkey:bigint:REGULAR
        Input: 1500000 rows (21.46MB), Filtered: 0.00%, Physical Input: 4.18MB
```

Click here to explore the query plan fragment.

Step 15 - Review plan Fragment 2

Review the labeled graphic of Fragment 2 below. Click on each plus sign to learn more about the corresponding section of the fragment.

Note: Don't forget to read from the bottom up.

```
Fragment 2 [SOURCE]
    CPU: 758.34ms, Scheduled: 1.41s, Blocked 5.55s (Input: 1.21s, Output: 0.00ns), Input: 1650000 rows (36.91MB); per task: avg.: 1650000.00 std.dev.: 0.00, Output: 6 rows
(144B)
    Output layout: [orderstatus, count_5, $hashvalue_11]
    Output partitioning: HASH [orderstatus][$hashvalue_11]
    Aggregate[type = PARTIAL, keys = [orderstatus], hash = [$hashvalue_11]]
    │   Layout: [orderstatus:varchar(1), $hashvalue_11:bigint, count_5:bigint]
    │   CPU: 46.00ms (4.88%), Scheduled: 46.00ms (1.68%), Blocked: 0.00ns (0.00%), Output: 6 rows (144B)
    │   Input avg.: 339727.00 rows, Input std.dev.: 92.78%
    │   Collisions avg.: 0.00 (0.00% est.), Collisions std.dev.: ?%
    │   count_5 := count(*)
    └─ Project[]
       │   Layout: [orderstatus:varchar(1), $hashvalue_11:bigint]
       │   Estimates: {rows: 1500000 (21.46MB), cpu: 21.46M, memory: 0B, network: 0B}
       │   CPU: 20.00ms (2.12%), Scheduled: 19.00ms (0.69%), Blocked: 0.00ns (0.00%), Output: 679454 rows (9.72MB)
       │   Input avg.: 339727.00 rows, Input std.dev.: 92.78%
       │   $hashvalue_11 := combine_hash(bigint '0', COALESCE("$operator$hash_code"("orderstatus"), 0))
       └─ InnerJoin[criteria = ("custkey" = "custkey_0"), hash = [$hashvalue_7, $hashvalue_8], distribution = REPLICATED]
          │   Layout: [orderstatus:varchar(1)]
          │   Estimates: {rows: 1500000 (8.58MB), cpu: 45.23M, memory: 34.33MB, network: 0B}
          │   CPU: 492.00ms (52.23%), Scheduled: 616.00ms (22.51%), Blocked: 3.12s (25.79%), Output: 679454 rows (3.89MB)
          │   Left (probe) Input avg.: 33991.00 rows, Input std.dev.: 92.74%
          │   Right (build) Input avg.: 750000.00 rows, Input std.dev.: 0.00%
          │   Collisions avg.: 8444.98 (4.05% est.), Collisions std.dev.: 6.30%
          │   Distribution: REPLICATED
          │   dynamicFilterAssignments = {custkey_0 -> #df_505}
          ├─ ScanFilterProject[table = lakehouse_query_plan:query_plan_with_stats:customer, filterPredicate = (round("acctbal") > 5E3), dynamicFilters = {"custkey" =
#df_505}]
          │      Layout: [custkey:bigint, $hashvalue_7:bigint]
          │      Estimates: {rows: 150000 (2.57MB), cpu: 2.57M, memory: 0B, network: 0B}/{rows: 135000 (2.32MB), cpu: 2.57M, memory: 0B, network: 0B}/{rows: 135000
(2.32MB), cpu: 2.32M, memory: 0B, network: 0B}
          │      CPU: 69.00ms (7.32%), Scheduled: 383.00ms (13.99%), Blocked: 0.00ns (0.00%), Output: 67982 rows (1.17MB)
          │      Input avg.: 75000.00 rows, Input std.dev.: 92.69%
          │      $hashvalue_7 := combine_hash(bigint '0', COALESCE("$operator$hash_code"("custkey"), 0))
          │      custkey := custkey:bigint:REGULAR
          │      acctbal := acctbal:double:REGULAR
          │      Input: 150000 rows (2.57MB), Filtered: 54.68%, Physical Input: 7.24MB
          │      Dynamic filters:
          │          - df_505, ALL, collection time=518.88ms
          └─ LocalExchange[partitioning = HASH, hashColumn = [$hashvalue_8], arguments = ["custkey_0"]]
             │   Layout: [custkey_0:bigint, orderstatus:varchar(1), $hashvalue_8:bigint]
             │   Estimates: {rows: 1500000 (34.33MB), cpu: 34.33M, memory: 0B, network: 0B}
             │   CPU: 106.00ms (11.25%), Scheduled: 214.00ms (7.82%), Blocked: 1.23s (10.17%), Output: 1500000 rows (34.33MB)
             │   Input avg.: 750000.00 rows, Input std.dev.: 8.32%
             └─ RemoteSource[sourceFragmentIds = [3]]
                    Layout: [custkey_0:bigint, orderstatus:varchar(1), $hashvalue_9:bigint]
                    CPU: 18.00ms (1.91%), Scheduled: 127.00ms (4.64%), Blocked: 1.21s (10.00%), Output: 1500000 rows (34.33MB)
                    Input avg.: 750000.00 rows, Input std.dev.: 8.32%
```

Click here to explore the query plan fragment.

END OF LAB EXERCISE

Access control

Lab 1: Creating and validating RBAC policies with Starburst Galaxy

Estimated completion time

- 30 minutes

Learning objectives

- In this lab, you will create multiple schemas and then assign appropriate role-based access policies to existing roles.

Prerequisites

- Student setup - Lab 1: Create student account

Activities

1. Prepare for the exercise
2. Create Human Resources schema
3. Create Marketing schema
4. Grant privileges to hr role
5. Verify hr role's privileges
6. Grant and verify privileges to mktg role

Step 1 - Prepare for the exercise

- Sign in and verify the `students` role is selected in the upper-right corner.
- Ensure the `aws-us-east-1-free` **Cluster** is reporting a **Status** of `Running`.
- In the **Query editor**, select `aws-us-east-1-free` in the cluster pulldown.

Note: *If you did not previously create a schema*, execute the following SQL statement. Make sure that you replace `yourname` with your actual name, or another identifier that you prefer. **Only use lowercase characters and numbers; no special characters or spaces.**

```
CREATE SCHEMA students.yourname;
```

Step 2 - Create Human Resources schema

Create a schema and associated tables for the Human Resources department and load them with test data.

```
CREATE SCHEMA students.yourname_hr;
```

```
USE students.yourname_hr;
```

```
CREATE TABLE depts (
   id int, name varchar, mgr_id int, avg_emp_rating decimal(5,3)
);
INSERT INTO depts VALUES
   (501, 'Engineering', 105, 4.0),
   (502, 'Sales', 102, 4.5);

CREATE TABLE emps (
   id int, name varchar, dept int, rating decimal(5,3)
);
INSERT INTO emps VALUES
   (101, 'Fred', 501, 4.0),
   (102, 'Susan', 502, 4.5),
   (103, 'Beth', 501, 5.0),
   (104, 'Margo', 502, 4.5),
   (105, 'John', 501, 3.0);
```

Query both tables to verify they contain test data.

Step 3 - Create Marketing schema

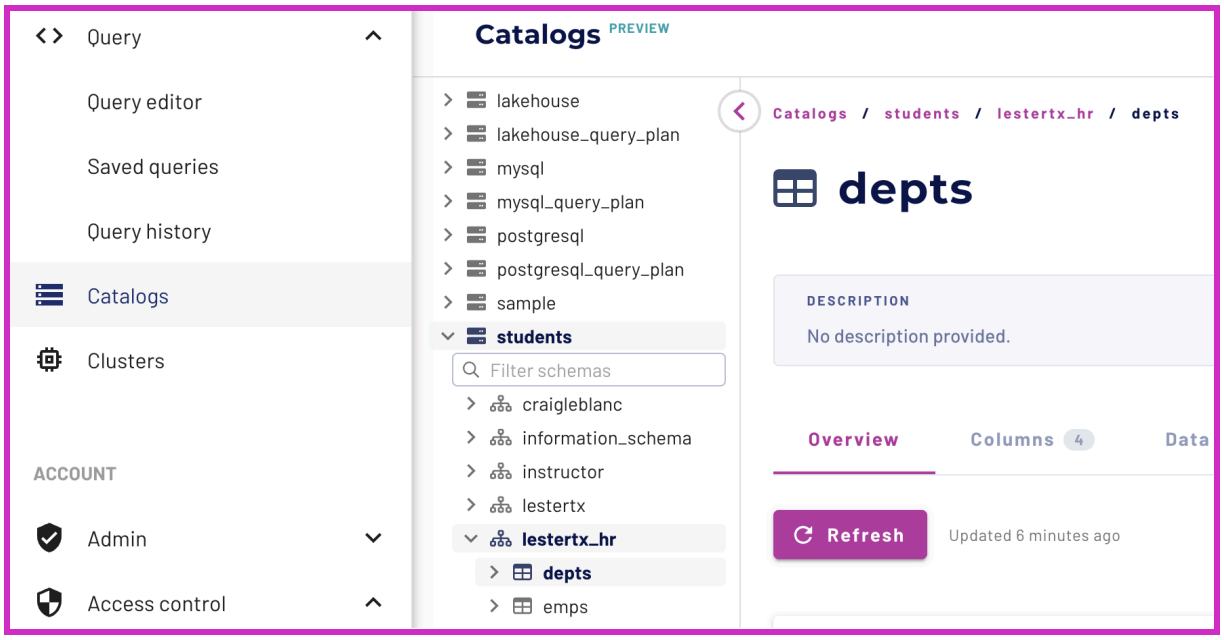
Create a schema and set of associated tables for the Marketing department. Once complete, load them with test data.

```
CREATE SCHEMA students.yourname_mktg;
USE students.yourname_mktg;

CREATE TABLE ext_campaign_results (
   id int, client_name varchar, bogus_results varchar
);
INSERT INTO ext_campaign_results VALUES
   (888, 'ABC Corp', 'nonsense nonsense'),
   (888, 'XYZ LLC', 'nonsense nonsense'),
   (999, 'XYX LLC', 'nonsense nonsense'),
   (999, '123 Inc', 'nonsense nonsense');

CREATE TABLE int_survey_responses (
   id int, dept int, bogus_response varchar
);
INSERT INTO int_survey_responses VALUES
   (1101, 502, 'stuff stuff stuff'),
   (1101, 501, 'stuff stuff stuff'),
   (1101, 502, 'stuff stuff stuff'),
   (2101, 501, 'stuff stuff stuff'),
   (2101, 502, 'stuff stuff stuff');
```

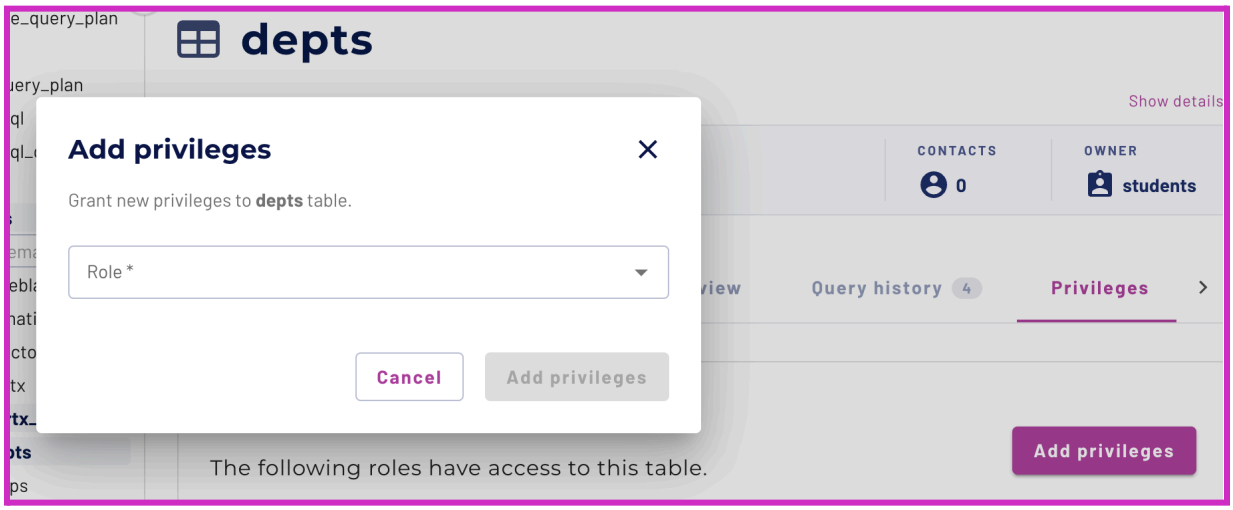
Query both tables to verify they contain test data.

Step 4 - Grant privileges to hr role

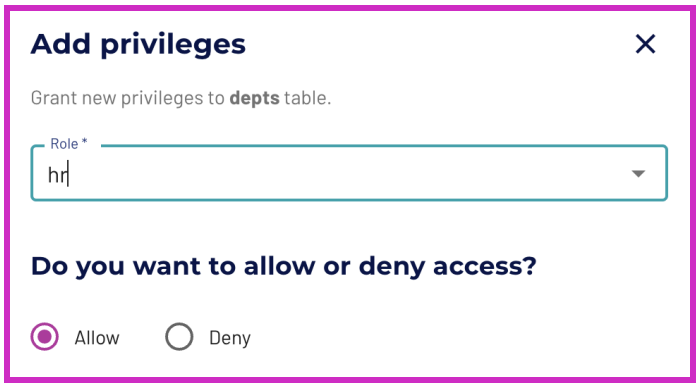
Select **Catalogs** in the left navigation. Under **Catalogs,** in the middle pane, expand `students` and then `yourname_hr`. Click on the `depts` table.



Click on the **Privileges** tab on the far right of the screen. Click on the **Add privileges** button that appears, which surfaces the **Add privileges** dialogue box.

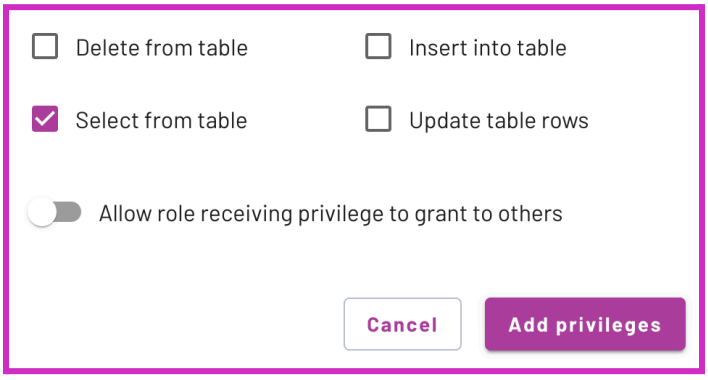


Choose the `hr` role in the pulldown control. Confirm that the default **Allow** radio button is selected.

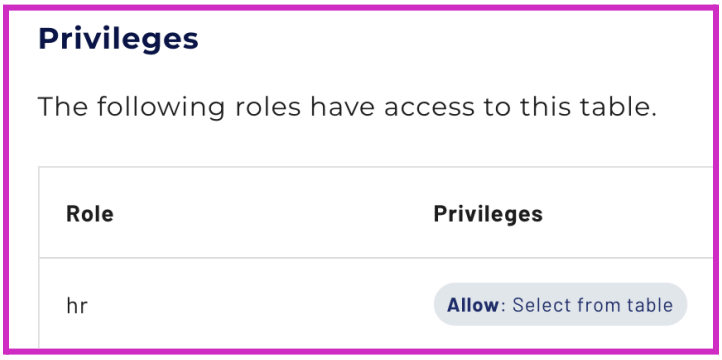


Now it's time to select the appropriate privileges. You want to enable select rights for this table.

Check the box labeled **Select from table** and then click the **Add privileges** button.



You will see a new entry in the **Privileges** table listing the `hr` role as **Select from table privileges**.

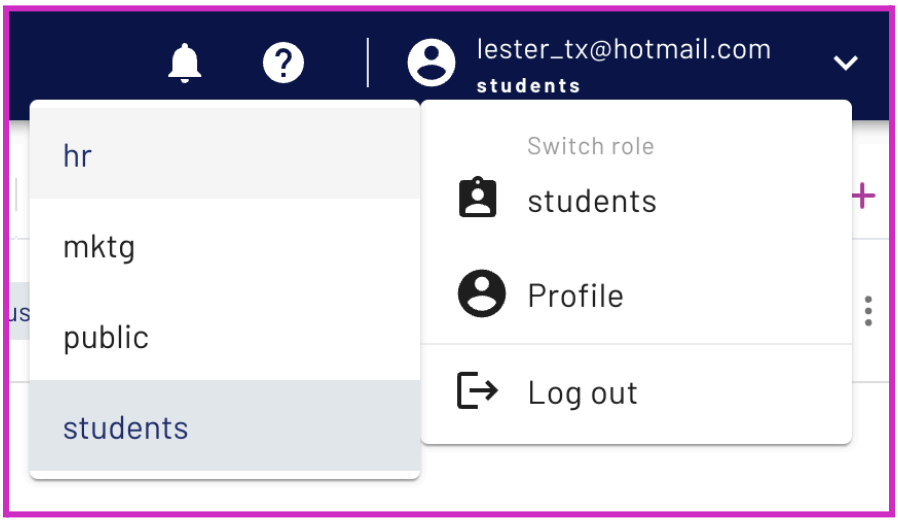


Using the same procedure, grant the same privilege to the `hr` role for the `emps` table.

Step 5 - Verify hr role's privileges

Return to the **Query editor** and expand the `aws-us-east-1-free` cluster. You will see multiple catalogs listed.

In the upper right corner of the UI, click on the down arrow next to your email address, and hover over **students** in the pull-down menu that is exposed. A submenu will surface as shown below and click on **hr**.

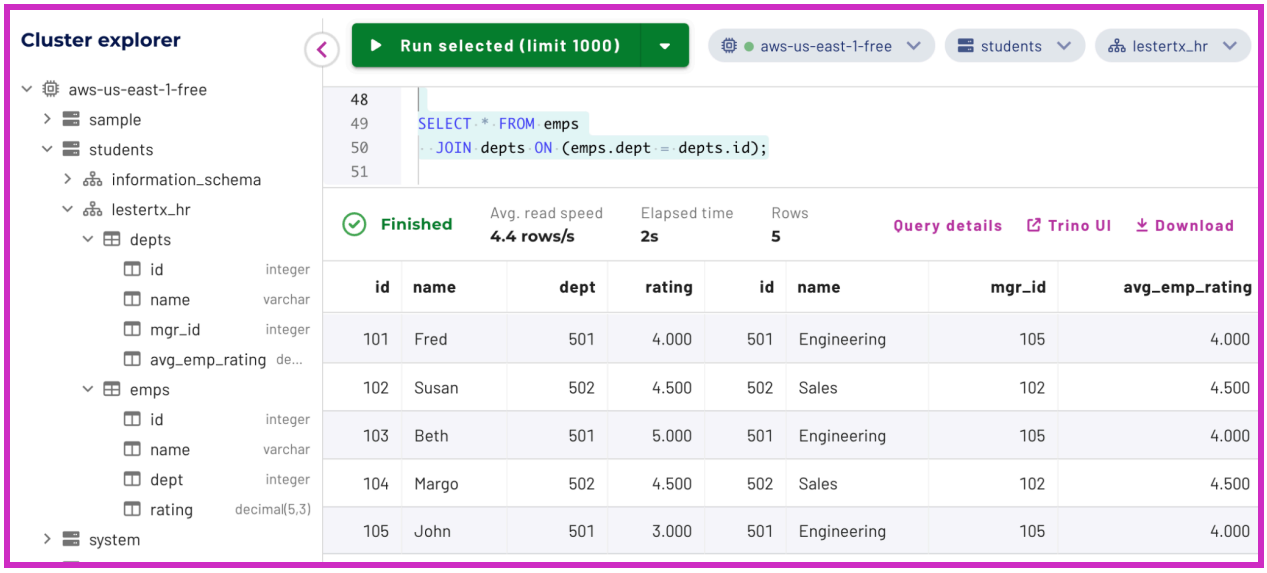


The upper right corner of the UI will now indicate your session has changed to the **hr** role.

The expandable cluster/catalog/schema list will now have the catalogs view collapsed. Drill back into the `aws-us-east-1-free` cluster and the `students` catalog. Beyond `information_schema`, you should only see the `yourname_hr` schema.

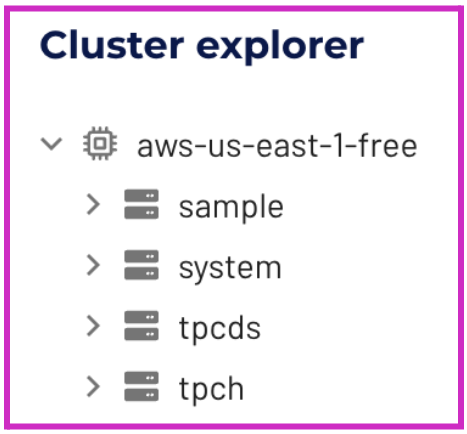
The `depts` and `emps` tables should be present as well. Verify that you can query both of them using the code below.

```
USE students.yourname_hr;

SELECT * FROM emps JOIN depts ON (emps.dept = depts.id);
```



Switch to the **mktg** role in the upper right corner.

Verify that the `students` schema is not present under the cluster tree.



Now, switch to the **students** role.

Step 6 - Grant and verify privileges to mktg role

Using **Steps 4 & 5** as a guide, repeat the steps to assign the two tables in the `yourname_mktg` schema to the `mktg` role. Then switch to the **mktg** role in the UI and verify both tables are accessible. Verify that the **hr** role cannot access them.

Switch back to the **mktg** role and execute the following query to obtain the HR department name for each record in the internal survey table using the logical foreign key that is present.

```
SELECT *
 FROM students.yourname_mktg.int_survey_responses AS s
 JOIN students.yourname_hr.depts AS d
   ON (s.dept = d.id);
```

As you probably expected, the following appropriate error surfaced.

> Access Denied: Cannot select from columns [id] in table or view students.lestertx_hr.depts: Role mktg does not have the privilege SELECT on the column [id]

To resolve this, switch back to the **students** role in the UI, then grant the `mktg` role privileges to run `SELECT` statements on the `depts` table. Verify that the privilege is working correctly by executing the query above again in the **mktg** role.

END OF LAB EXERCISE

Lab 2: Creating and validating ABAC policies with Starburst Galaxy

Estimated completion time

- 30 minutes

Learning objectives

- In this lab, you will apply existing tags, and their associated attribute-based access controls, to particular columns and then verify the these policies deny access to the specific columns.

Prerequisites

- Student setup - Lab 1: Create student account

Activities

1. Prepare for the exercise
2. Create Sales schema
3. Explore existing tags
4. Assign tags to `customer` table
5. Assign tag to `orders` table
6. Grant privileges to mktg role
7. Grant privileges to sales role
8. Understand existing policies on `pii` and `sales` tags
9. Verify sales role's privileges
10. Verify mktg role's privileges

Step 1 - Prepare for the exercise

- Sign in and verify the `students` role is selected in the upper-right corner.
- Ensure the `aws-us-east-1-free` **Cluster** is reporting a **Status** of `Running`.
- In the **Query editor**, select `aws-us-east-1-free` in the cluster pulldown.

Note: *If you did not previously create a schema,* execute the following SQL statement. Make sure that you replace `yourname` with your actual name, or another identifier that you prefer. **Only use lowercase characters and numbers; no special characters or spaces.**

```
CREATE SCHEMA students.yourname;
```

Step 2 - Create Sales schema

Create a schema and associated tables for the Sales department and load them with test data.

```
CREATE SCHEMA students.yourname_sales;
```

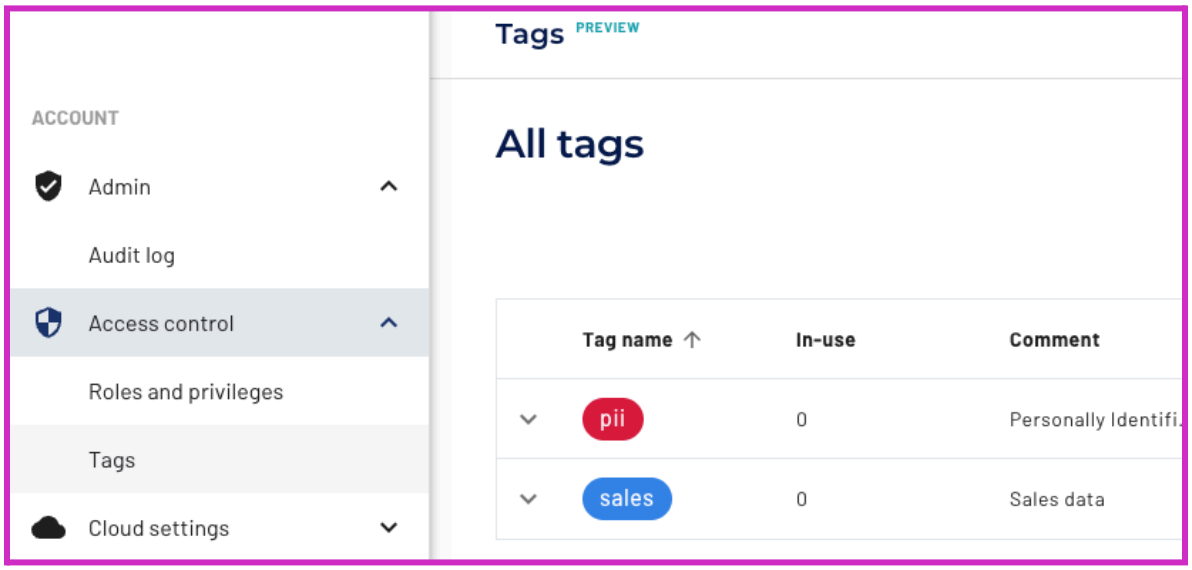
```
USE students.yourname_sales;
```

```
CREATE TABLE orders AS
SELECT * FROM tpch.tiny.orders;

CREATE TABLE customer AS
SELECT * FROM tpch.tiny.customer;
```

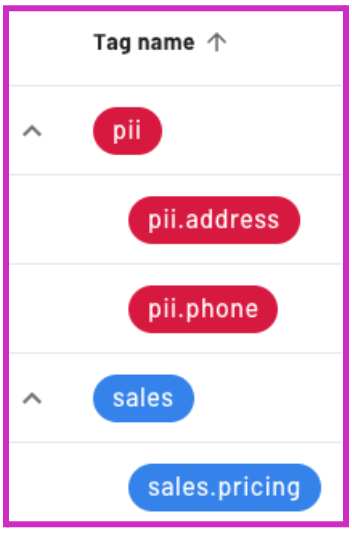
Query both tables to verify they contain test data.

Step 3 - Explore existing tags

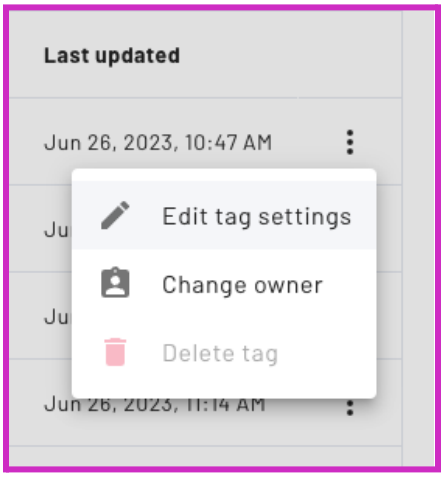
Select **Access control** and then **Tags** in the left navigation. The **All tags** page will render.



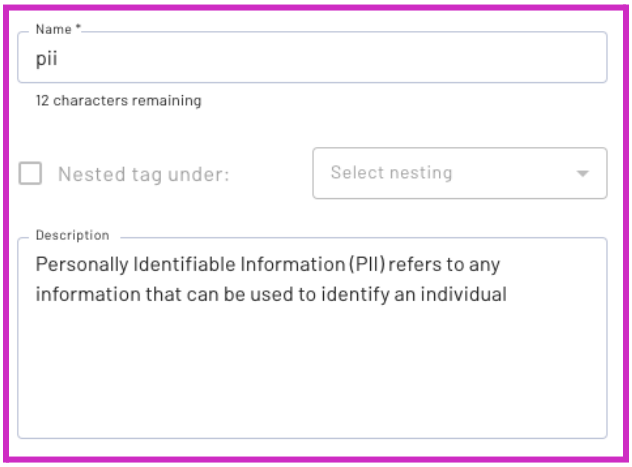
Click on the down arrow to the left of the **Tag name** items in the list to expand to see additional tabs that are nested.



The top-level tags of `pii` and `sales`, and their associated nested tags, will be used in this exercise. To see the full comment for each, scroll the table contents to the right and click on the vertical ellipses and then select **Edit tag settings**.

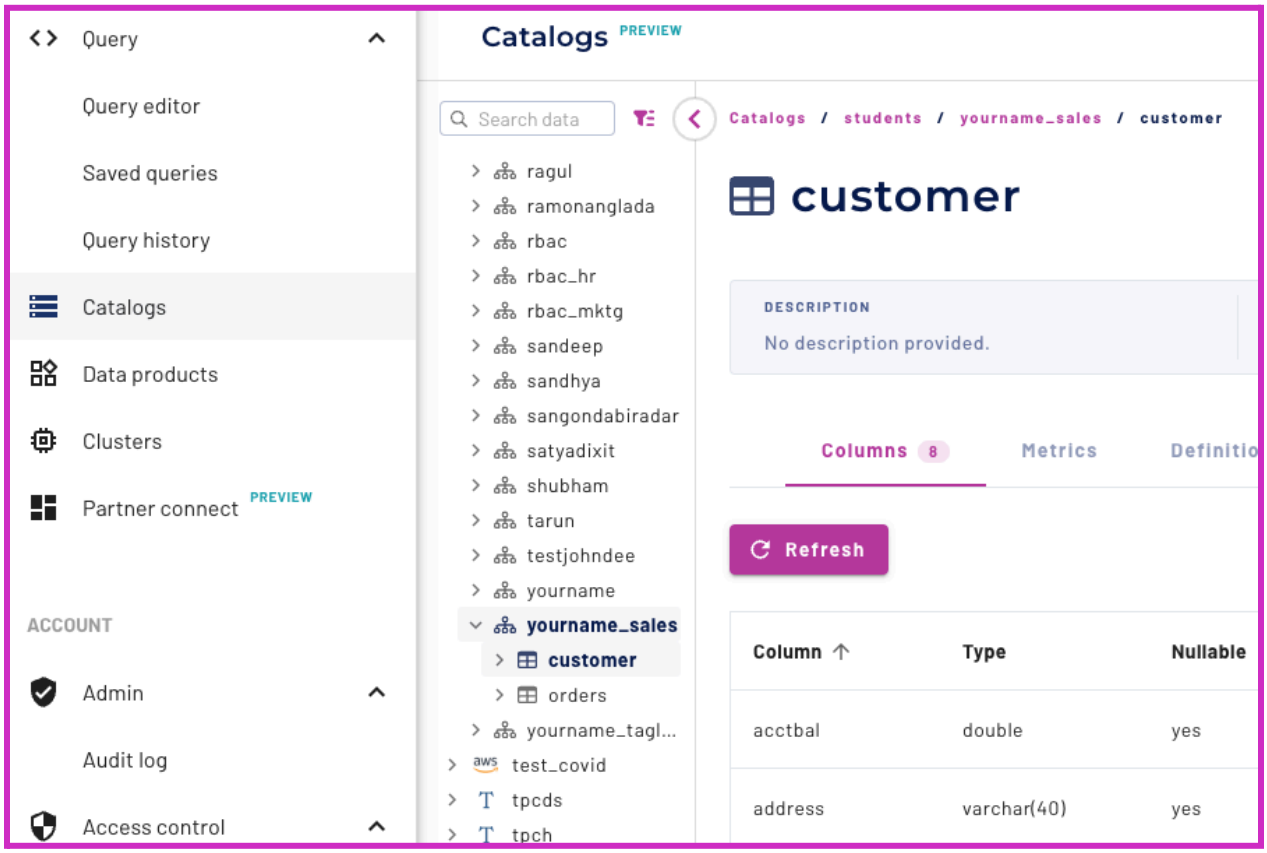


You can now see the full Description of the tag.



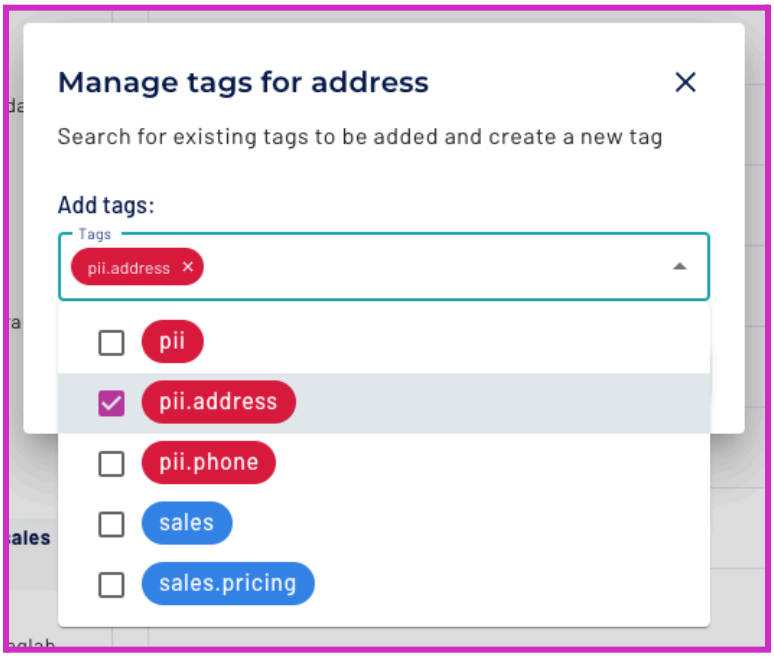
Step 4 - Assign tags to `customer` table

Select **Catalogs** in the left navigation. Under **Catalogs,** in the middle pane, expand `students` and then `yourname_sales`. Click on the `customer` table.

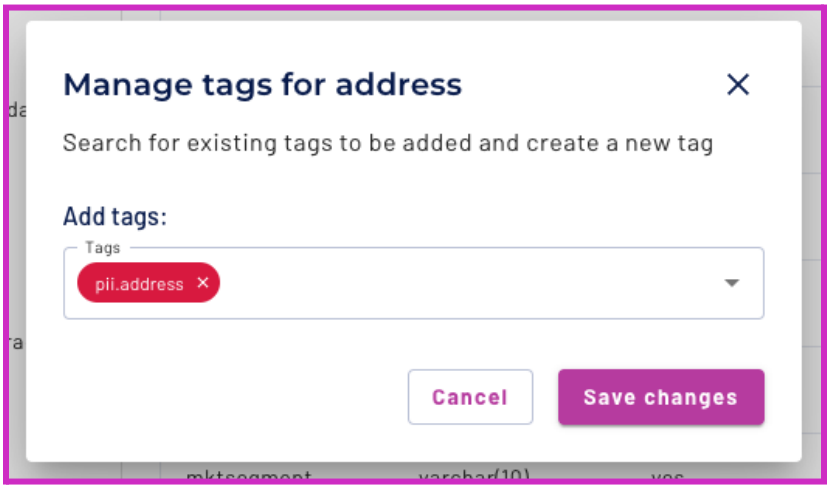


The **Columns** tab should be selected. Scroll down until you see the `address` column name and then look to the right to find the UI column titled **Tags**. Click the **+** next to **No tags assigned** as show in the far right in the following screenshot.

Expand the **Add tags** drop-down, and check the box in front of `pii.address`.



Click once outside the drop-down to close it. Press the **Save changes** button.



Notice that the tag is not assigned.

Using the same procedure, apply the `pii.phone` tag to the `phone` column.

Using the same procedure, apply the `pii` tag to the `name` column.

Confirm that all 3 tags were assigned correctly.

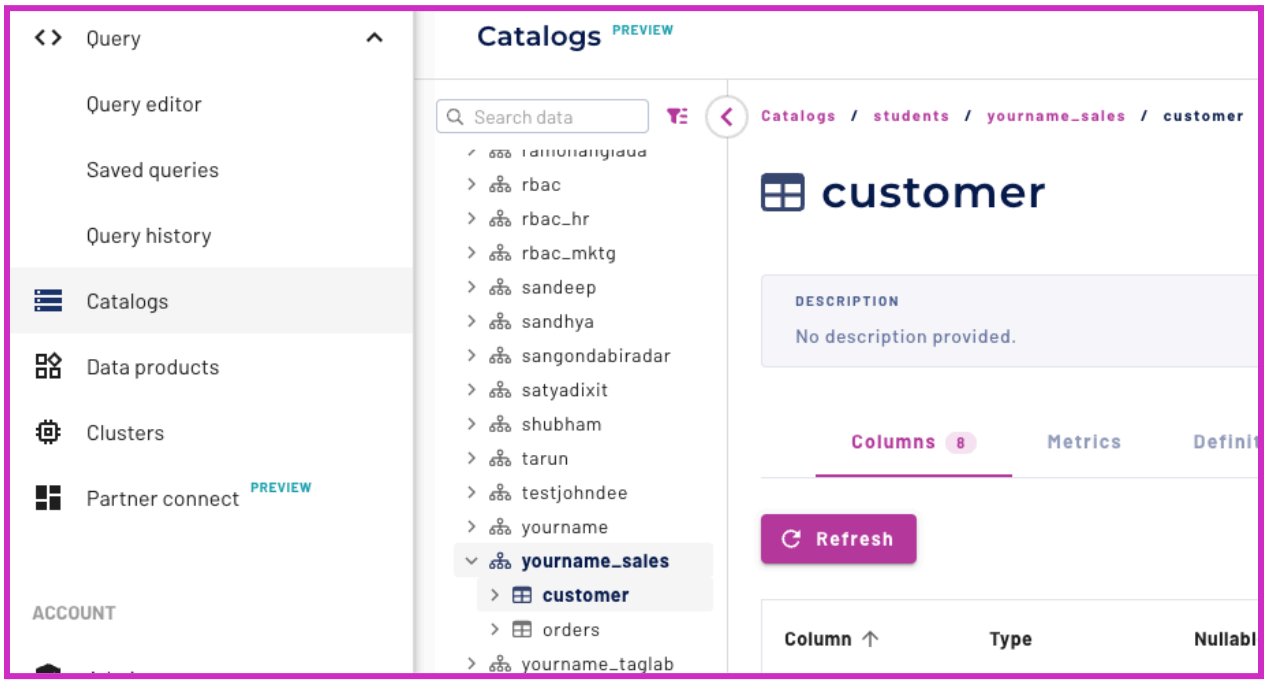
Column	Type	Nullable	Default	Tags
acctbal	double	yes	null	No tags assigned. +
address	varchar(40)	yes	null	pii.address +
comment	varchar(117)	yes	null	No tags assigned. +
custkey	bigint	yes	null	No tags assigned. +
mktsegment	varchar(10)	yes	null	No tags assigned. +
name	varchar(25)	yes	null	pii +
nationkey	bigint	yes	null	No tags assigned. +
phone	varchar(15)	yes	null	pii.phone +

Step 5 - Assign tag to `orders` table

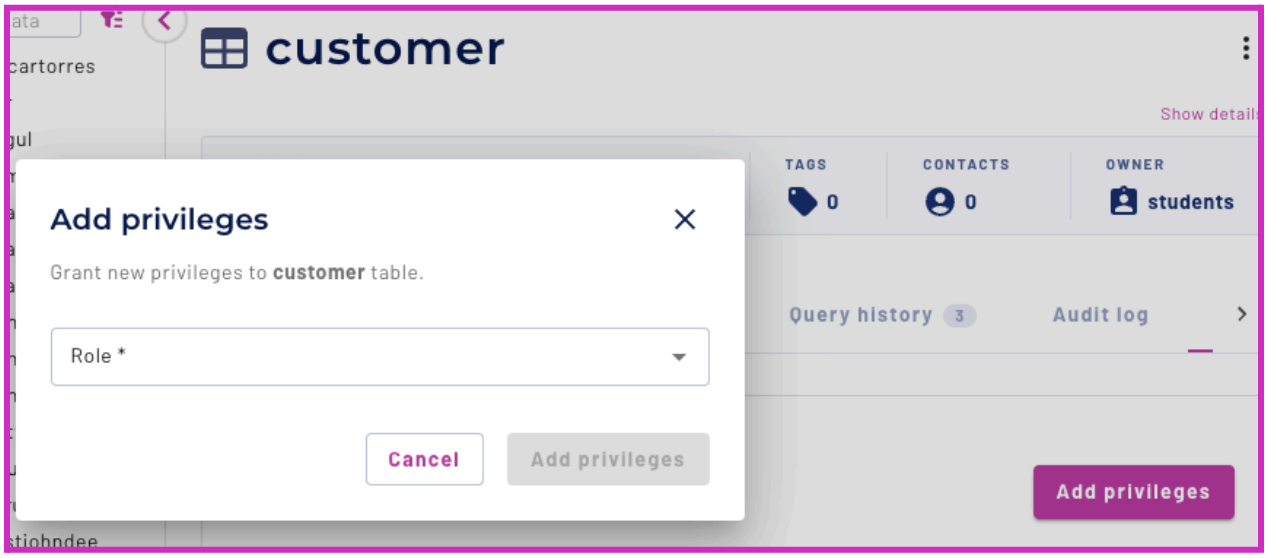
Using the process from **Step 4**, apply the `sales.pricing` tag to the `totalprice` column of the `orders` table. Confirm that the tag was assigned correctly.

Step 6 - Grant privileges to mktg role

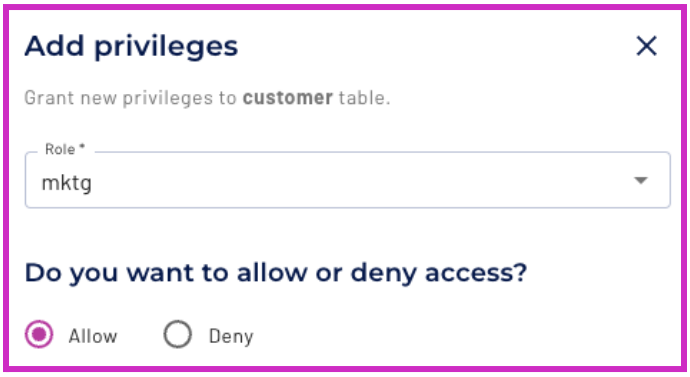
Select **Catalogs** in the left navigation. Under **Catalogs,** in the middle pane, expand `students` and then `yourname_sales`. Click on the `customer` table.



Click on the **Privileges** tab on the far right of the screen. Click on the **Add privileges** button that appears, which surfaces the **Add privileges** dialogue box.

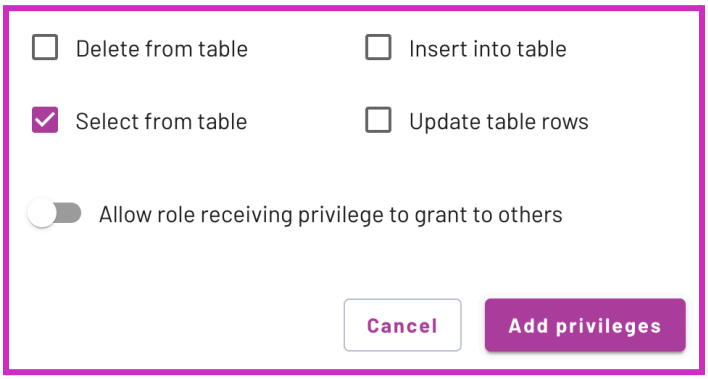


Choose the `mktg` role in the pulldown control. Confirm that the default **Allow** radio button is selected.

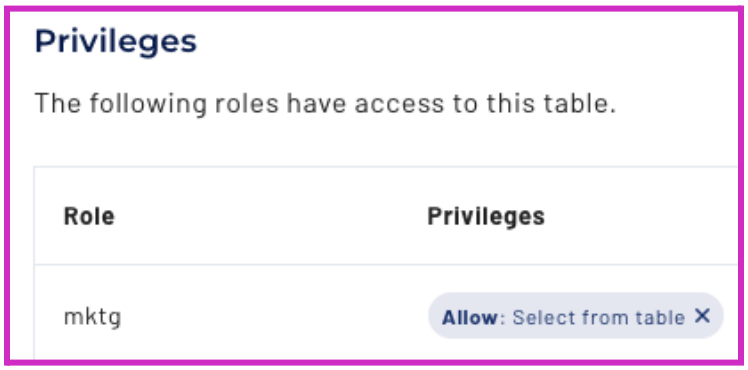


Now it's time to select the appropriate privileges. You want to enable select rights for this table.

Check the box labeled **Select from table** and then click the **Add privileges** button.



You will see a new entry in the **Privileges** table listing the `hr` role as **Select from table privileges**.



Using the same procedure, grant the same privilege to the `mktg` role for the `orders` table.

Step 7 - Grant privileges to sales role

Using the process from **Step 6**, grant the same privilege on the `customer` and `orders` tables to the `sales` role.

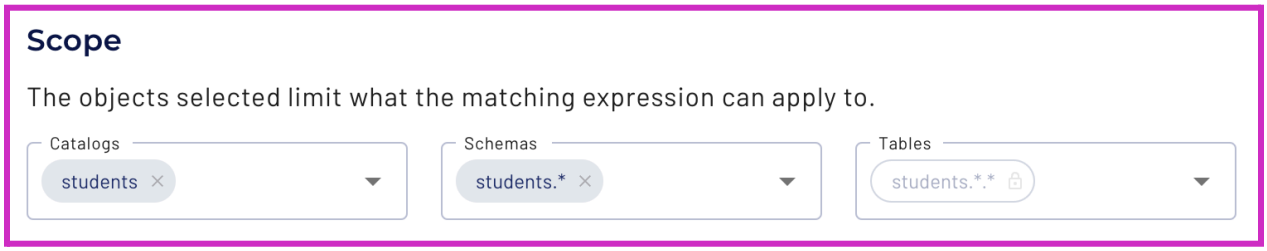
Step 8 - Understand existing policies on `pii` and `sales` tags

An administrator previously established some policies that help control access against the `pii` and `sales` tags you applied to your `customer` and `orders` table in **Step 4** and **Step 5**.

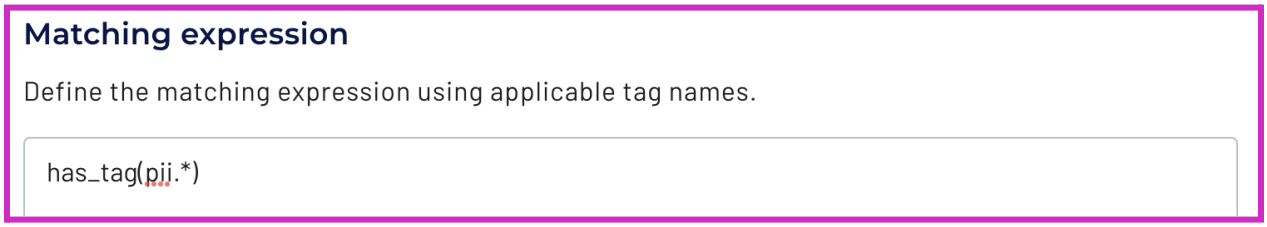
Note: this step is showing you what those configurations would look like if you had permissions to edit them.

An access control policy was created and named `deny_pii`.

It is applicable for the entire `students` catalog.



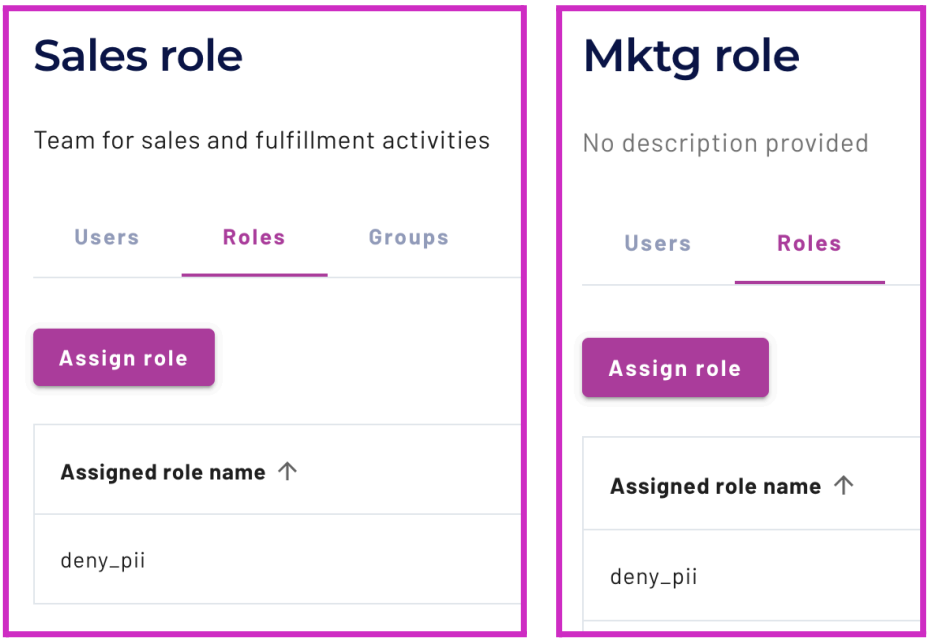
The policy is aligned to any tag that is nested under the pii tag.



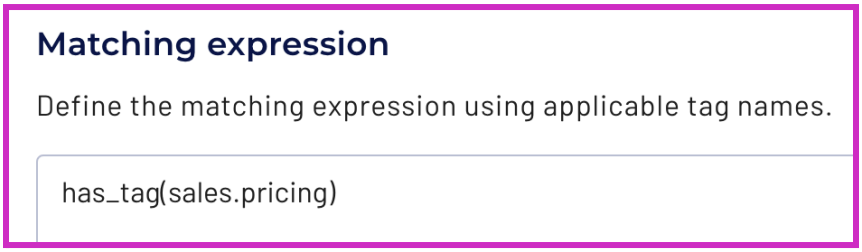
Note: `has_tag(pii.*)` will only match nested tags (ex: `pii.phone` or `pii.address`) and will not match the parent tag. Conversely, `has_tag(pii)` will only match the parent tag (ex: `pii`) and will not match any of the nested tags. If you wanted to match both the parent tag and the nested tags, you would use the following; `has_tag(pii) OR has_tag(pii.*)`.

The policy prevents read access to items that have tags aligned with the **Matching expression**.

This policy is bound to a role with the same name; `deny_pii`. That role has been added to the `sales` and `mktg` roles in order for the tag-based policy to be complete.

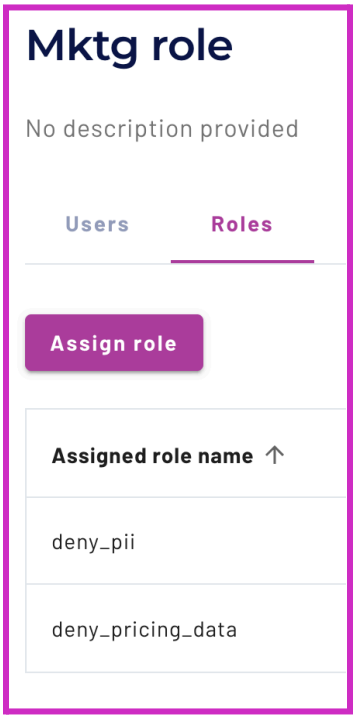


Another role with a similar policy, both named `deny_pricing_data`, is present that has the following **Matching expression**.



Based on the earlier explanation, this policy is configured to only consider the `pricing` tag that is nested under the `sales` tag. *Neither the parent tag, or any other nested tag, matches.*

This role has only beed added to the `mktg` role in addition to the earlier role.

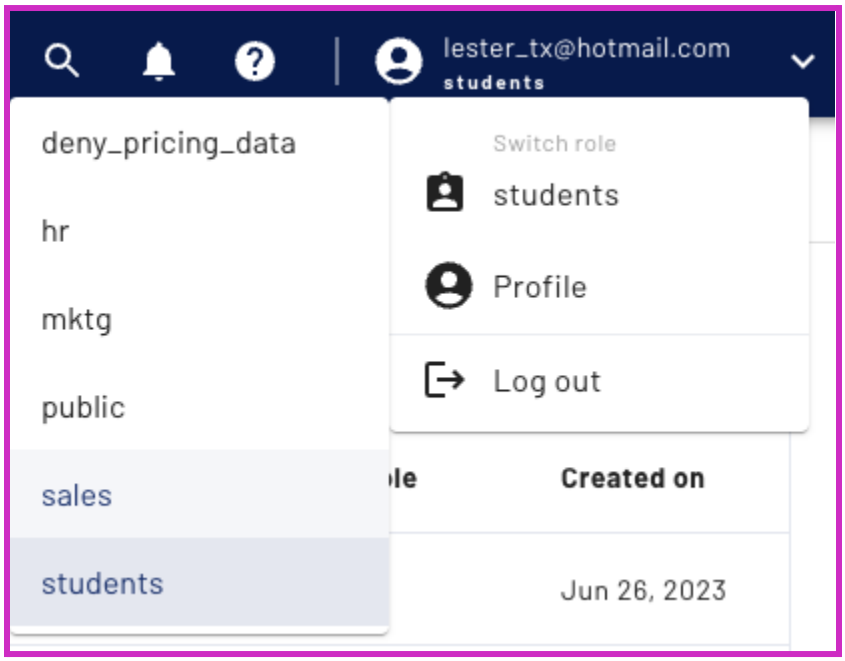


Step 9 - Verify sales role's privileges

The prior step declares that the `sales` role was denied access to the `address` and `phone` columns in the `customer` table. No policies were established against the `orders` table.

Return to the **Query editor** and expand the `aws-us-east-1-free` cluster. You will see multiple catalogs listed.

In the upper right corner of the UI, click on the down arrow next to your email address, and hover over **students** in the pull-down menu that is exposed. A submenu will surface as shown below and click on **sales**.

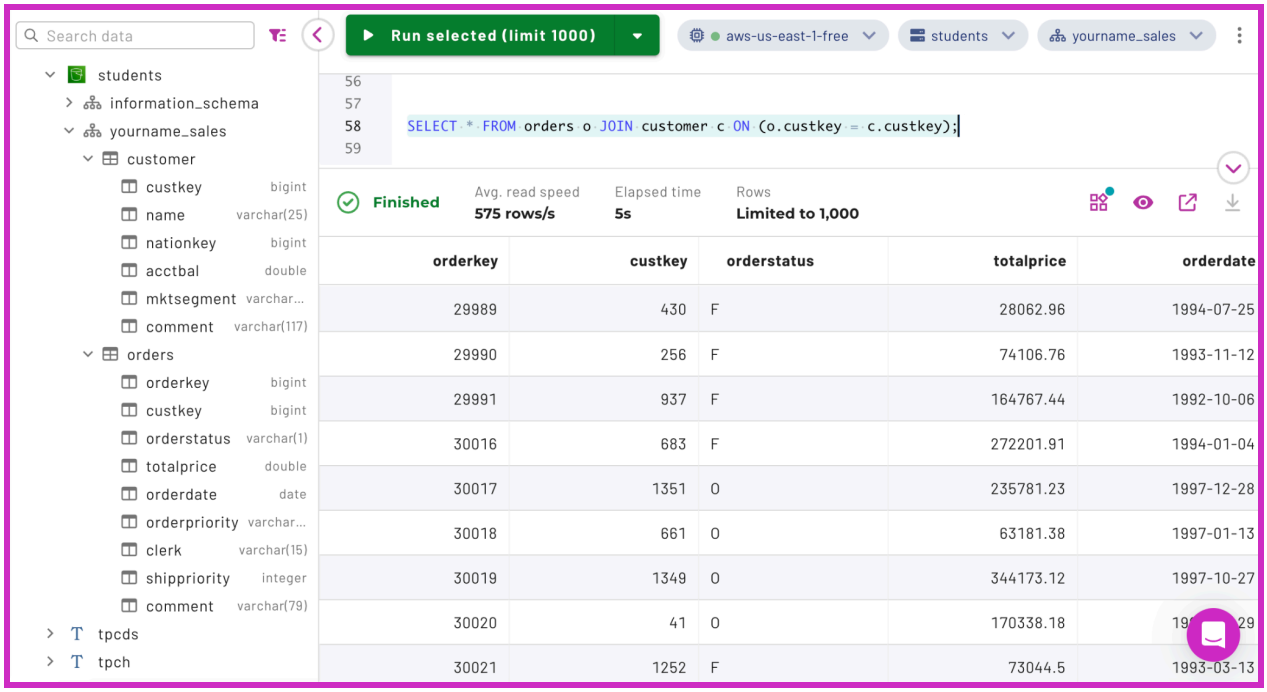


The upper right corner of the UI will now indicate your session has changed to the **sales** role.

The expandable cluster/catalog/schema list will now have the catalogs view collapsed. Drill back into the `aws-us-east-1-free` cluster and the `students` catalog. Expand the `yourname_sales` schema.

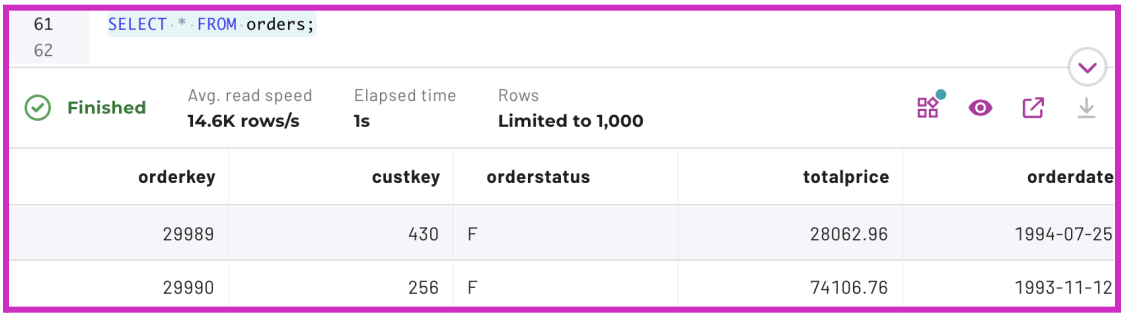
The `customer` and `orders` tables should be present. Verify that you can query both of them using the code below.

```
USE students.yourname_sales;

SELECT * FROM orders o JOIN customer c ON (o.custkey = c.custkey);
```

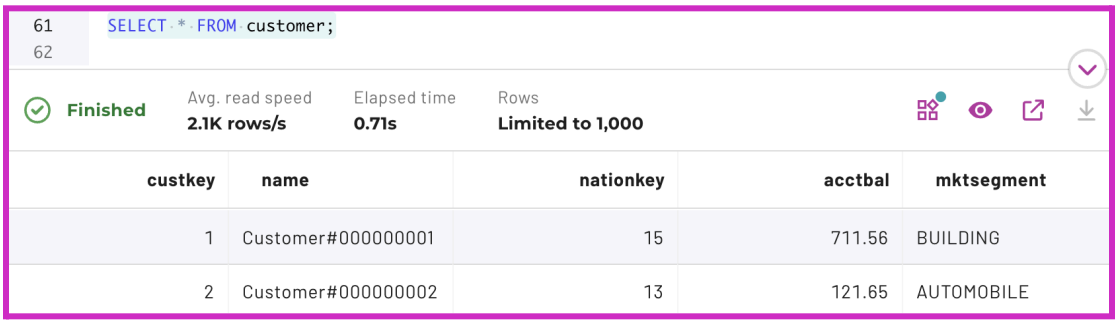



On the surface, this seem problematic since there should have been some restrictions on the `customer` table that prevented these results from being returned.

Run a simple `SELECT * FROM orders;` query to verify that it returns unimpeded.



Run the same thing against the `customer` table.



This seems wrong as some of the pii tagged columns should not be accessible. Reviewing the columns closely you will notice the `name` column is present, but the `phone` and `address` column are not. Even the table hierarchy in the middle UI pane does not show these two columns.

Note: when you have a `SELECT *` on a table and do not try to project specific columns, Starburst Galaxy omits those columns from the results that the user does not have access to.

Run the following query on just the pii related columns.

```
SELECT name, phone, address FROM customer;
```

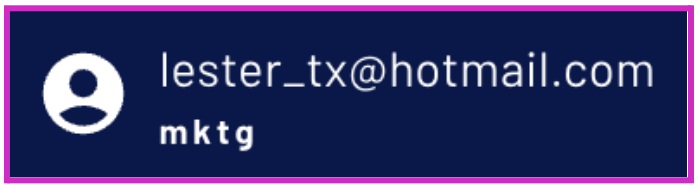
The error message identifies exactly which columns are the problem; `address` and `phone`. The `name` column is not an issue as it was tagged with the base `pii` tag, not one of its nested tags.

> Access Denied: Cannot select from columns [address, phone, name] in table or view students.yourname_sales.customer: Role sales does not have the privilege SELECT on the columns [address, phone]

Step 10 - Verify mktg role's privileges

Just as with the `sales` role, the `mktg` role was denied access to the `address` and `phone` columns in the `customer` table. Additionally, it was denied access to the `totalprice` column in the `orders` table.

In the upper right corner of the UI, click on the down arrow next to your email address, and change the role selector to use **mktg**.



The expandable cluster/catalog/schema list will now have the catalogs view collapsed. Drill back into the `aws-us-east-1-free` cluster and the `students` catalog. Expand the `yourname_sales` schema.

The `customer` and `orders` tables should be present. Verify that the results from these 3 queries are the same as for the `sales` role from **Step 9**.

```
SELECT * FROM orders o JOIN customer c ON (o.custkey = c.custkey);
SELECT * FROM customer;
SELECT name, phone, address FROM customer;
```

Verify the `mktg` role's restriction on the `sales.pricing` tagged column, `totalprice`, does not cause an error, and is ignored, when `*` is used for the columns to return.

```
SELECT * FROM orders;
```

Verify it is a problem when the column is called out specifically.

```
SELECT orderkey, totalprice FROM orders;
```

The error message identifies the problem with the `totalprice` as expected.

> Access Denied: Cannot select from columns [orderkey, totalprice] in table or view students.yourname_sales.orders: Role mktg does not have the privilege SELECT on the column [totalprice]

END OF LAB EXERCISE

Data pipelines

Lab 1: Construct a pipeline with insert-only transactions

Estimated completion time

- 60 minutes

Learning objectives

- In this lab, you will gain an appreciation for the complexities involved in creating a full data pipeline that spans the Land, Structure, and Consume layers of the data lakehouse. The implementation you will construct involves a variety of methods of data pipelining, including batch, insert-only, and a full data pipeline.

Prerequisites

- Student setup - Lab 1: Create student account

Activities

1. Prepare for the exercise
2. Review pipeline dataset
3. Review pipeline requirements
4. Review initial load dataset
5. Simulate initial ingestion to Land layer
6. Transform data types
7. Enrich the data
8. Perform initial load to Structure layer
9. Create views for Consume layer
10. Execute the pipeline for an incremental ingest
11. Perform the incremental ingest again
12. Create plan for automation

Step 1 - Prepare for the exercise

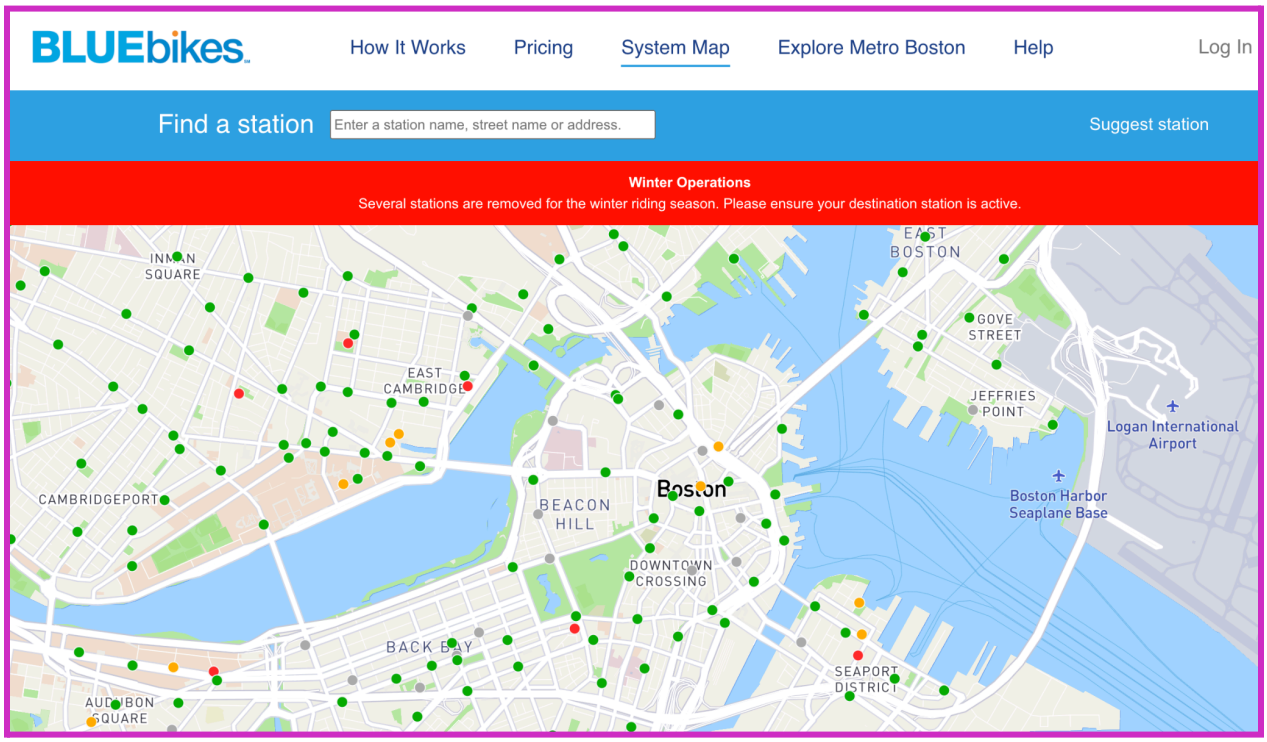
- Sign in and verify the `students` role is selected in the upper-right corner.
- Ensure the `aws-us-east-1-free` **Cluster** is reporting a **Status** of `Running`.
- In the **Query editor**, select `aws-us-east-1-free` in the cluster pulldown.

Note: *If you did not previously create a schema*, execute the following SQL statement. Make sure that you replace `yourname` with your actual name, or another identifier that you prefer. **Only use lowercase characters and numbers; no special characters or spaces.**

```
CREATE SCHEMA students.yourname;
```

Step 2 - Review pipeline dataset

For this lab, you will use the publicly available Bluebikes - Hubway dataset. Read more information about Blue Bikes Boston, a bicycle-sharing program based in Boston since 2011.



This time, you will be focused on the transactional records of the bike trips from start to finish.

Step 3 - Review pipeline requirements

Creating a data pipeline requires several layers, also known as zones. Each layer will be created separately and will serve a different purpose in the overall pipeline ecosystem.

Review the following high-level requirements to ensure that you understand this reference architecture and each of its three layers before proceeding.

Land layer

- Creates an ingestion process for the bike trips data, which is:
    - Triggered by the creation of a new dataset.
    - Lands raw data into a working table.
- The land layer defines a location on the data lake to store historical raw data

Structure layer

- Transforms data in the Land layer to produce the new, complete, row-level dataset.
- Its objective should be:

- Ensuring data is of high quality
- Perform casting of datatypes to most the appropriate type.
- Augment the records by adding the following columns based on the rider's postal code. These include:
  - Province (state for the USA).
  - Average income.

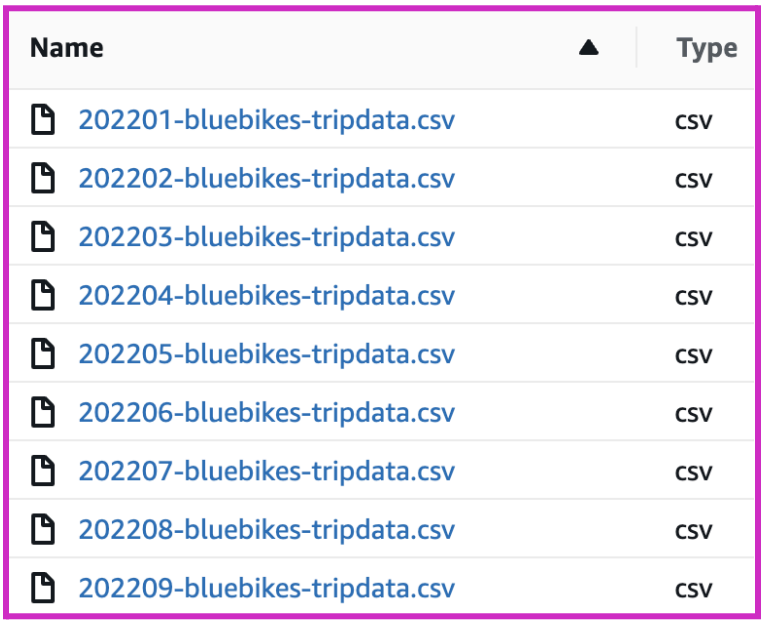
Consume layer

- Build a view that calculates minimum, maximum, and average duration by starting station and user type across all rides
- Build a view that calculates hour of day average income based on rider postal code broken out by month

Step 4 - Review initial load dataset

The initial bulk load of trip records is focused on January - September 2022. There is no attempt to build the ingestion process at this time, but it can be simulated. Although you do not have access to S3, there is a directory in the data lake.

It contains a single file for each of the nine months we are performing our one-time sync-up load.



Optionally, you could download one, or more, of these datasets as a zip file by visiting the download page at https://s3.amazonaws.com/hubway-data/index.html, but the lab instructions will use the data already identified above.

Here are a few records from the first file in the list above.

Note: To aid in readability and to avoid line-wrapping of these wide records, the "\" character is used to indicate the next line of indented text is still part of the record that started in the first column.

```
"tripduration","starttime","stoptime", \
   "start station id","start station name", \
   "start station latitude","start station longitude", \
   "end station id","end station name", \
   "end station latitude","end station longitude", \
   "bikeid","usertype","postal code"
597,"2022-01-01 00:00:25.1660","2022-01-01 00:10:22.1920", \
   178,"MIT Pacific St at Purrington St", \
   42.35957320109044,-71.10129475593567, \
   74,"Harvard Square at Mass Ave/ Dunster", \
   42.373268,-71.118579, \
   4923,"Subscriber","02139"
411,"2022-01-01 00:00:40.4300","2022-01-01 00:07:32.1980", \
   189,"Kendall T",42.362427842912396,-71.08495473861694, \
   178,"MIT Pacific St at Purrington St", \
   42.35957320109044,-71.10129475593567, \
   3112,"Subscriber","02139"
476,"2022-01-01 00:00:54.8180","2022-01-01 00:08:51.6680", \
   94,"Main St at Austin St",42.375603,-71.064608, \
   356,"Charlestown Navy Yard", \
   42.374124549426526,-71.05481199938367, \
   6901,"Customer","02124"
```

To make sure you can read the input data correctly, verify that the second ride record lasted 411 seconds, started at station 189, ended at station 178, and used bike 3112.

Step 5 - Simulate initial ingestion to Land layer

As this data is ingested as text files, we will use the Hive connector as it supports more file formats than the more modern table formats such as Iceberg. More specifically, we are ingesting comma-separated values for each record. We will use the `CSV` file type that can handle the quoted-fields seen in the sample data.

A consequence of this file type is that we can only use the `varchar` datatype for each column. Fortunately, this works well with our strategy of keeping all raw data exactly as received and also allows potentially low-quality, or even invalid, data to be stored by avoiding issues that are applied to other, more restrictive, data types.

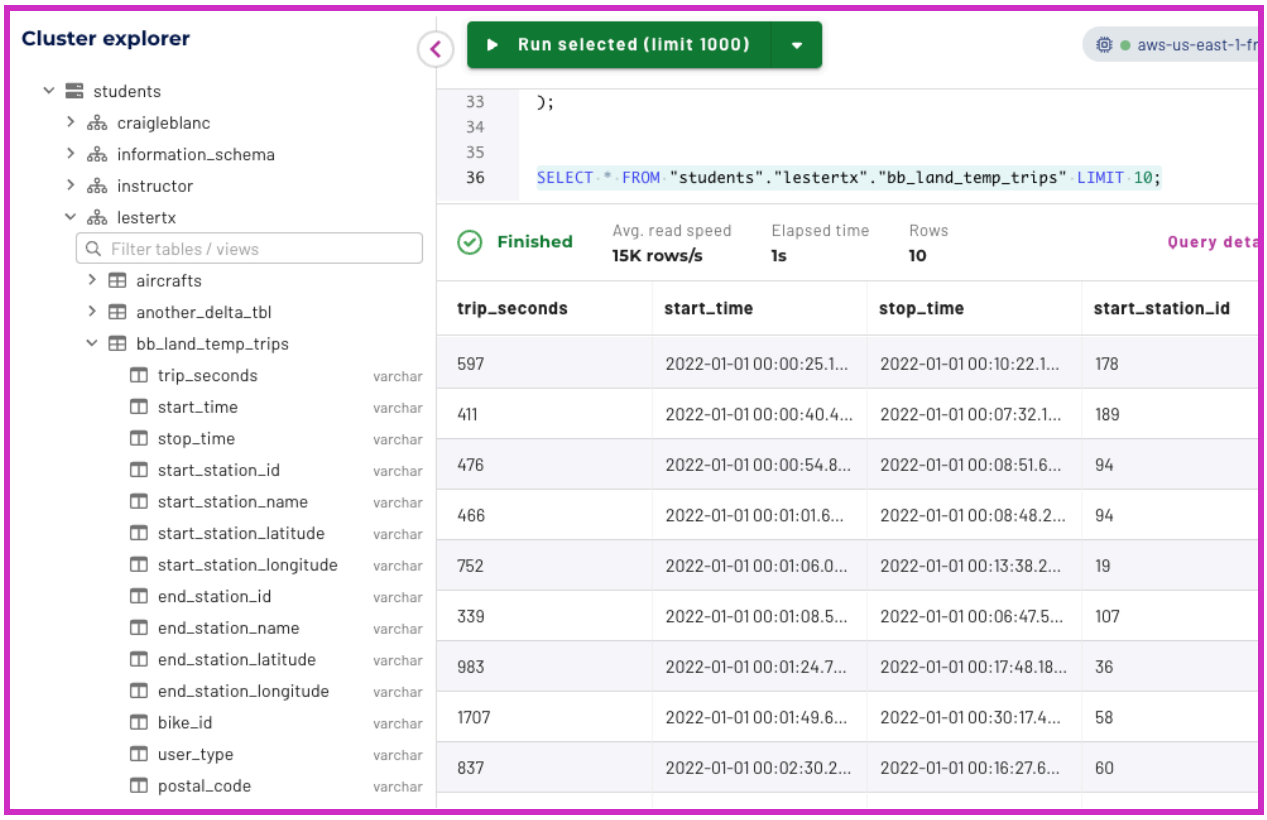
Create an ingest table in our logical Land layer to temporarily house the data being ingested.

Note: In a production environment, significant effort would have gone into establishing an appropriate `catalog.schema.table` naming convention. For this lab, all tables will be created in your schema within the `students` catalog. For example, the `bluebikes.land.temp_ingest_trips` table. The tables created in this lab will be an abbreviated form of this possible naming scheme.


```
USE students.yourname;

CREATE TABLE bb_land_temp_trips (
 trip_seconds varchar,
 start_time varchar,
 stop_time varchar,
 start_station_id varchar,
 start_station_name varchar,
 start_station_latitude varchar,
 start_station_longitude varchar,
 end_station_id varchar,
 end_station_name varchar,
 end_station_latitude varchar,
 end_station_longitude varchar,
 bike_id varchar,
 user_type varchar,
 postal_code varchar
) WITH (
 type = 'HIVE', format = 'CSV', csv_separator = ',',
 csv_quote = '"', skip_header_line_count = 1,
 --EXTERNAL table as ingestion process landed data
 external_location =
's3://edu-train-galaxy/bluebikes/raw_trips-2022_01-2022-09/'
);
```

List the newly created table's columns and inspect the data contained in it.

```
SELECT * FROM bb_land_temp_trips LIMIT 10;
```



The directory used for the ingestion process corresponds to the external table's location on the data lake. New data will land in this directory. As we will see later, the underlying data files will be moved into a final resting location within the Land layer along with all previously-ingested data after it has been transformed into the Structure layer.

Note: Normally, the `external_location` property would be a consistent directory, but for future ingestion data, you will recreate the table referencing another location. This is happening as we are not fully automating the end-to-end data pipeline and will not move the temporary ingestion data into its final location.

Step 6 - Transform data types

All of the columns in the `bb_land_temp_trips` table are of data type `varchar`. In practice, a data engineer would work through all columns to check which data type would be most appropriate. They would then perform some inspection of the data present to verify if any particular validity checks need to be implemented and/or if any transformations are required.

For example, the following query's results suggest the trip_seconds column could easily be placed into a numeric datatype as the existing `varchar` column's boundaries appear to be valid.

```
SELECT min(trip_seconds), max(trip_seconds)
  FROM bb_land_temp_trips;
```

Based on that observation, it appears that we could simply cast the column to use an integer datatype. While working on column by column, create a `SELECT` statement that keeps growing with the next column until you address all of them.

Note: At this point, there is no need to store the results during this exploration effort until after we address all columns.

```
SELECT CAST(trip_seconds AS int) AS trip_seconds
  FROM bb_land_temp_trips;
```

The next two columns represent specific timestamp values. Try to perform a similar cast on both of them using the code below.

Note: Remember to add these to the ongoing query.

```
SELECT
   CAST(trip_seconds AS int) AS trip_seconds,
   CAST(start_time AS timestamp(6)) AS start_time,
   CAST(stop_time AS timestamp(6)) AS stop_time
FROM bb_land_temp_trips;
```

Note: Typically, it is considered best practice to conduct deeper analysis to verify that the values are casting correctly to the `timestamp` datatype. However, for this lab, this check is unnecessary.

You can also simulate that you did the proper due diligence and found out that all of the columns, except for the last, were being received with high quality and could be transformed into their appropriate column types. You will see the full set of data type conversions just below. For now, focus on the last column, `postal_code`.

```
SELECT min(postal_code), max(postal_code)
  FROM bb_land_temp_trips;
```

The results look like this.

_col0	_col1
	Y1A 2P1

What does it mean? For starters, the value on the right indicates there are Canadian postal codes present. If you explore further, you will find the majority of the values are USA postal codes. There are possibly postal codes from even more countries. There are also some empty strings. Are there more than just a few?

```
SELECT count()
  FROM bb_land_temp_trips
 WHERE postal_code = '';
```

Are there any values that start with a space?

```
SELECT count()
  FROM bb_land_temp_trips
 WHERE postal_code LIKE ' %';
```

Upon further analysis, assume you determined there are some records that are intended to indicate the field was null.

```
SELECT count()
  FROM bb_land_temp_trips
 WHERE postal_code = 'NULL';
```

This can be fixed using the two functions together. The innermost function below is “replace” ; it will replace NULL with an empty string (i.e. ''). The outermost function below is “nullif” ; it will take all of the empty strings and set them to a true “NULL” value. Since the innermost function runs first this means that both the original and new empty stringsare all set to NULL.

```
SELECT
  nullif(replace(postal_code, 'NULL', ''), '')
FROM bb_land_temp_trips;
```

Putting all of the data type conversion activities together, you will have this query. Notice that there are a few columns where no conversions are needed, such as `user_type`.

```
SELECT
CAST(trip_seconds AS int) AS trip_seconds,
CAST(start_time AS timestamp(6)) AS start_time,
CAST(stop_time AS timestamp(6)) AS stop_time,
CAST(start_station_id AS int) AS start_station_id,
start_station_name,
CAST(start_station_latitude AS decimal(15,13))
  AS start_station_latitude,
CAST(start_station_longitude AS decimal(16,13))
  AS start_station_longitude,
CAST(end_station_id AS int) AS end_station_id,
end_station_name,
CAST(end_station_latitude AS decimal(15,13))
  AS end_station_latitude,
CAST(end_station_longitude AS decimal(16,13))
  AS end_station_longitude,
CAST(bike_id AS int) AS bike_id,
user_type,
nullif(replace(postal_code, 'NULL', ''), '')
  AS postal_code
FROM bb_land_temp_trips;
```

This was a simplified version of what this initial activity might require depending on the number of columns present and the quality of the data being ingested.

Step 7 - Enrich the data

The enrichment requirements declared the records need to be augmented by adding province and average income values based on rider postal code. As the following query shows, there is an existing lookup table that can be utilized. As the table name suggests, it only contains USA-based zip codes. Here are the columns we can leverage for our enrichment needs.


```
SELECT zip_code, state,
       format_number(tot_inc_avg * 1000) AS avg_income
  FROM lakehouse.global.zip_code_income
 WHERE zip_code IN
     (30004, 30009, 30022, 30075, 30076, 30092);
```

The zip codes used above are for Roswell, Georgia, USA. If you know enough about a specific USA zip code, run this query again to see if the results are similar to what you are expecting. If you do not have a zip code to try, use `90210`, which is Beverly Hills, California, USA.

Verify that the two postal_code values present in Step 3's sample data are present in the zip code lookup table.

```
SELECT * FROM lakehouse.global.zip_code_income
 WHERE zip_code IN (02139, 02124);
```

Notice anything about the results for these two USA zip codes?

state	zip_code
MA	2124
MA	2139

The `zip_code` column is of type `INT`, and thus the leading zero is missing in the results. Verify that these two values will join correctly based on the fact that the `bb_land_temp_trips.postal_code` is, and will remain, a `varchar` datatype.

```
SELECT t.postal_code, z.tot_inc_amt
 FROM bb_land_temp_trips AS t
 JOIN lakehouse.global.zip_code_income AS z ON (
   cast(t.postal_code as INT) = z.zip_code)
WHERE postal_code IN ('02139', '02124');
```

To expedite the lab exercise, assume in your analysis of the data to be ingested you identified the following potential concerns

- The inability to cast the `postal_code` column to the `INT` data type due to these conditions that are present
  - Canadian postal codes and possibly other unknown string formats
  - Empty strings
  - Strings with a hard-coded value of `NULL`
  - Values with a dash in them indicating the USA's ZIP+4 format
- Values in `postal_code` column that are not in the `zip_code_income` table, including
  - `99999`
  - `00000`

Further assume that you were able to produce the following SQL that identifies how to augment the ingestion data with the state name and average income from the `zip_code_income` table.

```
SELECT CAST(t.trip_seconds AS int) AS trip_seconds,
       nullif(replace(t.postal_code, 'NULL', ''), '') AS postal_code,
       cast(z.state AS VARCHAR(2)) as province,
       z.tot_inc_avg * 1000 AS avg_income
  FROM bb_land_temp_trips AS t
  LEFT JOIN lakehouse.global.zip_code_income AS z
    ON (try_cast(split_part(nullif(replace(t.postal_code, 'NULL',
''), ''), '-', 1)  AS int) = z.zip_code);
```

Your results should be similar to these.

trip_seconds	postal_code	province	avg_income
455	02115	MA	103530.12380000
369	02134	MA	42662.79548000
447	02142	MA	177228.47680000
329	02144	MA	71786.85315000
416	11375	NY	85221.23016000

For any value that cannot be joined on the lookup table, null values will be used for the augmented fields.

Step 8 - Perform initial load to Structure layer

Build a single transformation query by melding together the data conversions/casting efforts along with the enrichment activities. As this is our initial load of the Structure layer table, you can use the Create Table As Select (CTAS) approach to build the table to store this sync-up effort. This table will be used to store high-quality in future instances of this data pipeline you are constructing.

```
CREATE TABLE bb_structure_trips
 WITH (type='iceberg', format='orc') AS (
SELECT
CAST(t.trip_seconds AS int) AS trip_seconds,
CAST(start_time AS timestamp(6)) AS start_time,
CAST(stop_time AS timestamp(6)) AS stop_time,
CAST(t.start_station_id AS int) AS start_station_id, t.start_station_name,
CAST(t.start_station_latitude AS decimal(15,13)) AS start_station_latitude,
CAST(t.start_station_longitude AS decimal(16,13)) AS start_station_longitude,
CAST(t.end_station_id AS int) AS end_station_id, end_station_name,
CAST(t.end_station_latitude AS decimal(15,13)) AS end_station_latitude,
CAST(t.end_station_longitude AS decimal(16,13)) AS end_station_longitude,
CAST(t.bike_id AS int) AS bike_id, t.user_type,
nullif(replace(t.postal_code, 'NULL', ''), '') AS postal_code,
cast(z.state AS VARCHAR(2)) as province,
cast(round(z.tot_inc_avg * 1000) AS INT) AS avg_income
FROM
 bb_land_temp_trips AS t
 LEFT JOIN lakehouse.global.zip_code_income AS z ON (
   try_cast(
     split_part(
       nullif(replace(t.postal_code, 'NULL', ''), ''), '-', 1
     ) AS int
   ) = z.zip_code
 )
);
```

After a cursory review of the new table, validate that the temporary ingestion table has the same number of records as the newly constructed Structure table.

```
SELECT count() FROM bb_land_temp_trips;
SELECT count() FROM bb_structure_trips;
```

They both should identify 2,906,784 rows present.

To assist the Cost-based optimizer in creating the best possible query plan, calculate statistics on this new table.

```
ANALYZE bb_structure_trips;
```

You created an Iceberg table with the ORC file format as you want this Structure layer table to be performance-oriented and transaction capable.

At this point in the pipeline, you would normally move the underlying data lake files from the `external_location` of the `bb_land_temp_trips` table to the data lake folder backing the table that will keep all raw data within. That table is not being created in this lab as you do not have access to the S3 object store to perform the necessary file moves.

Note: With the naming scheme we have been using, that table would be called `bb_land_raw_trips` in your schema. A possible production name might be `bluebikes.land.raw_trips`.

Step 9 - Create views for Consume layer

As presented in **Step 2**, the following are the requirements for datasets in the Consume layer:

- Build a view that calculates minimum, maximum, and average duration by starting station and user type across all rides
- Build a view that calculates hour of day average income based on rider postal code broken out by month

The following SQL creates a view to support the first reporting requirement.

```
CREATE VIEW
bb_consume_dur_aggs_by_strt_stat_and_usr_typ
AS (
SELECT start_station_id, user_type,
       min(trip_seconds) AS min_trip_seconds,
       max(trip_seconds) AS max_trip_seconds,
       round(avg(trip_seconds)) AS avg_trip_seconds
  FROM bb_structure_trips
 GROUP BY start_station_id, user_type
);
```

Perform a cursory check to verify this view is functional. Additionally, here is a potential consumption of this view.

```
SELECT *
  FROM bb_consume_dur_aggs_by_strt_stat_and_usr_typ
 ORDER BY avg_trip_seconds DESC, start_station_id;
```

The following SQL creates a view to support the second reporting requirement.

```
CREATE VIEW
bb_consume_avg_inc_by_month_and_hour
AS
WITH dates_decomposed AS (
 SELECT substr(cast(date_trunc('month', start_time) AS varchar),
          1,  7) AS trip_month,
        substr(cast(date_trunc('hour',  start_time) AS varchar),
          12, 2) AS hr_of_day,
        avg_income AS avg_inc_from_with
   FROM bb_structure_trips
)
SELECT trip_month, hr_of_day,
      round(avg(avg_inc_from_with)) AS avg_income
 FROM dates_decomposed
GROUP BY trip_month, hr_of_day
;
```

Perform a cursory check to verify this view is functional. Additionally, here is a potential consumption of this view.

```
SELECT *
  FROM bb_consume_avg_inc_by_month_and_hour
 WHERE trip_month = '2022-04'
 ORDER BY hr_of_day;
```

Step 10 - Execute the pipeline for an incremental ingest

As identified in **Step 4**, the pipeline's ingestion process is being simulated. To accomplish this, drop the temporary ingest table and recreate it with a different `external_location` value pointing to the first incremental ingest data for October, 2022.

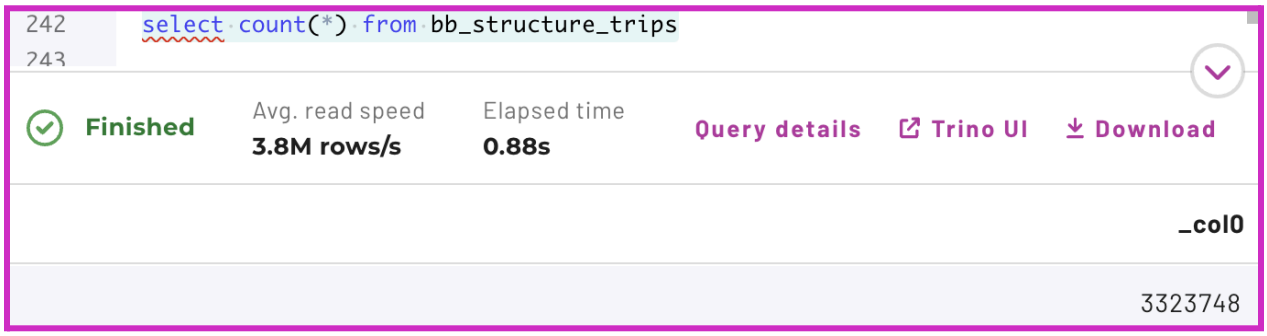
```
DROP TABLE bb_land_temp_trips;
CREATE TABLE bb_land_temp_trips (
trip_seconds varchar,
start_time varchar,
stop_time varchar,
start_station_id varchar,
start_station_name varchar,
start_station_latitude varchar,
start_station_longitude varchar,
end_station_id varchar,
end_station_name varchar,
end_station_latitude varchar,
end_station_longitude varchar,
bike_id varchar,
user_type varchar,
postal_code varchar
) WITH (
  type = 'HIVE', format = 'CSV', csv_separator = ',', csv_quote =
'"',
       skip_header_line_count = 1,
       --NEW data lake location
  external_location =
's3://edu-train-galaxy/bluebikes/raw_trips-2022_10/'
);
```

Perform a cursory check on this revised table. Validate that there are 416,964 rides in the `bb_land_temp_trips` table.

Instead of using a CTAS statement, you will use an INSERT statement in the Structure layer table using the same transformations you have used before.

```
INSERT INTO bb_structure_trips
SELECT
CAST(t.trip_seconds AS int) AS trip_seconds,
CAST(start_time AS timestamp(6)) AS start_time,
CAST(stop_time AS timestamp(6)) AS stop_time,
CAST(t.start_station_id AS int) AS start_station_id,
t.start_station_name,
CAST(t.start_station_latitude AS decimal(15,13))
  AS start_station_latitude,
CAST(t.start_station_longitude AS decimal(16,13))
  AS start_station_longitude,
CAST(t.end_station_id AS int) AS end_station_id,
end_station_name,
CAST(t.end_station_latitude AS decimal(15,13))
  AS end_station_latitude,
CAST(t.end_station_longitude AS decimal(16,13))
  AS end_station_longitude,
CAST(t.bike_id AS int) AS bike_id, t.user_type,
nullif(replace(t.postal_code, 'NULL', ''), '')
  AS postal_code,
cast(z.state AS VARCHAR(2)) as province,
cast(round(z.tot_inc_avg * 1000) AS INT) AS avg_income
FROM bb_land_temp_trips AS t
 LEFT JOIN lakehouse.global.zip_code_income AS z ON (
   try_cast(
     split_part(
       nullif(replace(t.postal_code, 'NULL', ''), ''), '-', 1
     ) AS int
   ) = z.zip_code
 );
```

Verify that the Structure layer table, which had 2,906,784 records previously, now has 416,964 more.



The next step would be to move the temporary ingestion table's underlying files to the location of the raw table with all historical information. Again, you cannot perform this step as you do not have access directly to S3.

Validate that the Consume layer has been updated with this new data.

```
SELECT *
  FROM bb_consume_avg_inc_by_month_and_hour
 WHERE trip_month = '2022-10'
 ORDER BY hr_of_day;
```

Step 11 - Perform the incremental ingest again

Repeat the activities from **Step 10**:

1. To emulate the ingestion process landing files in the `external_location` of the `bb_land_temp_trips` table
    a. Drop the table
    b. Create it again using the S3 folder ending with `raw_trips-2022_11/` for the `external_location` property
2. Run the `INSERT` statement into the Structure table from the Land table just recreated
3. Validate there are 3,614,369 total number of records present in the Structure table
4. Verify the Consume view(s) reflect this new data

Step 12 - Create plan for automation

At this point, you have validated the data pipeline three times. Once with the initial bulk load and then two more times performing incremental loads. You execute the steps of this batch ingestion pipeline manually.

An appropriate next step would be to automate and schedule the entire pipeline so it can run unattended. Tools such as Apache Airflow and dbt are possible tools to leverage in this effort.

Automating this pipeline is outside the scope of lab exercise, but in production, this would be a critical step to complete.

END OF LAB EXERCISE

Lab 2: Construct a pipeline with the `MERGE` statement

Estimated completion time

- 30 minutes

Learning objectives

- In this lab, you will construct the building blocks of a data pipeline whose ingestion data represents new and updated records needing to be updated in a Consume layer table. The implementation you will construct is a batch, merged-based data pipeline. It will utilize the `MERGE` statement as it allows us to not only address new records with a transaction, but also can tackle the updates and deletes of existing records in the same transaction.

Prerequisites

- Student setup - Lab 1: Create student account
- Apache Iceberg - Lab 3: Data modifications and snapshots with Iceberg
- Advanced Iceberg - Lab 1: Utilize Iceberg's MERGE statement

Activities

1. Prepare for the exercise
2. Review the pipeline dataset
3. Review pipeline requirements
4. Review initial load dataset
5. Simulate initial ingestion to Land layer
6. Transform data types
7. Perform initial load to Structure layer
8. Simulate ingestion of delta records
9. Build the pipeline's `MERGE` statement
10. Verify the `MERGE` logic
11. Next steps

Step 1 - Prepare for the exercise

- Sign in and verify the `students` role is selected in the upper-right corner.
- Ensure the `aws-us-east-1-free` **Cluster** is reporting a **Status** of `Running`.
- In the **Query editor**, select `aws-us-east-1-free` in the cluster pulldown.

Note: *If you did not previously create a schema*, execute the following SQL statement. Make sure that you replace `yourname` with your actual name, or another identifier that you prefer. **Only use lowercase characters and numbers; no special characters or spaces.**

```
CREATE SCHEMA students.yourname;
```

Step 2 - Review pipeline dataset

For this lab, you will use the publicly available Bluebikes - Hubway dataset. Read more information about Blue Bikes Boston, a bicycle-sharing program based in Boston since 2011.

More specifically, you will be focused on the Bluebikes stations, which are the places where the bikes are stored, and rides begin and end as described at https://www.bluebikes.com/system-data.

Step 3 - Review pipeline requirements

Create a data pipeline that will be executed manually with the following high-level requirements broken down by the three layers of the reference architecture.

Land layer

- Creates an ingestion process for the bike trips data, which is:
    - Triggered by the creation of a new dataset.
    - Lands raw data into a working table.
- The land layer defines a location on the data lake to store historical raw data

Structure layer

- Transforms data in the Land layer to produce the new, complete, row-level single source of truth for the dataset.
- Its objective should be:
    - Ensuring data is of high quality
    - Perform casting of datatypes to most the appropriate type.
    - Augment the records by adding the following columns based on the rider's postal code. These include:
        - Province (state for the USA).
        - Average income.

Step 4 - Review initial load dataset

The initial load of station records is from a version dated January 15, 2022. There is no attempt to build the ingestion process at this time, but it can be simulated. Although you can not access S3, there is a directory in the data lake.

It contains a single file based on the as-of-date identified in the folder above and will be used to establish the first version of this updateable table.

Optionally, you could download the current dataset as from https://s3.amazonaws.com/hubway-data/current_bluebikes_stations.csv, but the lab instructions will use the data already identified above.

Here are a few records from the file above. You will notice there is an extra header line identifying the date the data file was updated. That will be leveraged later in the exercise.

```
Last Updated,1/15/2022,,,,,,
Number,Name,Lat,Long,District,Public,Total docks,Deployment Year
K32015,1200 Beacon St,42.34414899,-71.11467361,Brookline,Yes,15,2021
W32006,160 Arsenal,42.36466403,-71.17569387,Watertown,Yes,11,2021
A32019,175 N Harvard St,42.363796,-71.129164,Boston,Yes,17,2014
```

To make sure you can read the input data correctly, verify that the second station number is `W32006` and that it is in the `Watertown` district.

Step 5 - Simulate initial ingestion to Land layer

As this data is ingested as text files, we will use the Hive connector as it supports more file formats than the more modern table formats such as Iceberg. More specifically, we are ingesting comma-separated values for each record. Assume that we have determined there are no quoted fields in the file and that you have determined that the data will be received with a high degree of quality that will allow it to be directly referenced by the most appropriate data types.

Knowing this, the `TEXTFILE` format is an appropriate choice as it does not force all columns to use the `varchar` datatype. Overall, this will allow the transformations centered around data quality to be less extensive.

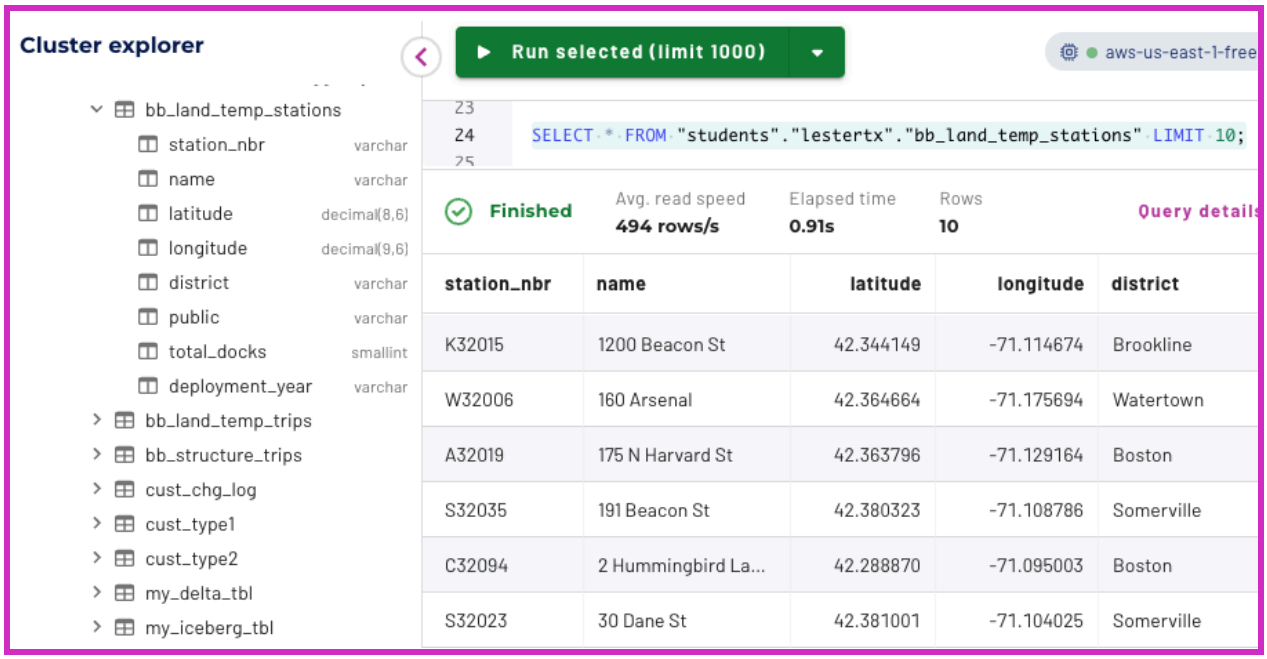
Create an ingest table in our logical Land layer to temporarily house the data being ingested.

Note: In a production environment, significant effort would have gone into establishing an appropriate `catalog.schema.table` naming convention. For this lab, all tables will be created in your schema within the `students` catalog. A possible example for this specific table might be `bluebikes.land.temp_ingest_stations` table. The tables created in this lab will be an abbreviated form of this possible naming scheme.

```
USE students.yourname;

CREATE TABLE bb_land_temp_stations (
  station_nbr varchar,
  name varchar,
  latitude decimal(8,6),
  longitude decimal(9,6),
  district varchar,
  public varchar,
  total_docks int,
  deployment_year varchar
) WITH (
 type = 'HIVE',
 --allow use of more than varchar
 format = 'TEXTFILE',
 textfile_field_separator = ',',
 --our files have TWO header rows to ignore
 skip_header_line_count = 2,
 --EXTERNAL table as ingestion process landed data
 external_location =
's3://edu-train-galaxy/bluebikes/raw_stations-2022-01-15/'
);
```

List the newly created table's columns and perform a cursory glance at the data contained in it.

```
SELECT * FROM bb_land_temp_stations;
```



This external table's location on the data lake is the directory into which the ingestion process will land new data. As we will see later, the underlying data files will be moved into a final resting location within the Land layer along with all previously ingested data after it has been transformed into the Structure layer.

Note: Normally, the `external_location` property would be a consistent directory, but for future ingestion data, you will recreate the table referencing another location. This is happening as we are not fully automating the end-to-end data pipeline and will not move the temporary ingestion data into its final location.

Step 6 - Transform data types

In practice, a data engineer would work through all columns to check for which datatype would be most appropriate. Then they would perform some inspection of the data present to verify if any particular validity checks need to be implemented and/or if any transformations that are required.

Assume that your research only yielded the following concerns.

- The district column has some empty string values that would be better represented as null values.
- The public column using Yes to indicate a positive value and a boolean data type would be more appropriate.
- The deployment_year column uses N/A in place of null.

Validate the following query resolves those issues.


```
SELECT
   station_nbr, name, latitude, longitude,
   nullif(district, '') AS district,
   starts_with(public, 'Yes') AS public,
   total_docks,
   try_cast(
       IF(
           deployment_year = 'N/A',
           'NULL',
           deployment_year
       ) AS int
   ) AS deployment_year
FROM
   bb_land_temp_stations;
```

Step 7 - Perform initial load to Structure layer

Build a single transformation query by melding together the data conversions/casting efforts along with the enrichment activities. As this is our initial load of the Structure layer table, you can use the Create Table As Select (CTAS) approach to build the table to store this sync-up effort. This table will be used to store high-quality in future instances of this data pipeline you are constructing.

```
CREATE TABLE bb_structure_stations
WITH (type='iceberg', format='orc') AS (
SELECT
   station_nbr, name, latitude, longitude,
   nullif(district, '') AS district,
   starts_with(public, 'Yes') AS public,
   total_docks,
   try_cast(
       IF(
           deployment_year = 'N/A',
           'NULL',
           deployment_year
       ) AS int
   ) AS deployment_year
FROM
   bb_land_temp_stations
);
```

After a cursory review of the new table, validate that the temporary ingestion table has the same number of records as the newly constructed Structure table.

```
SELECT count() FROM bb_land_temp_stations;
SELECT count() FROM bb_structure_stations;
```

They both should identify 448 rows present.

To assist the Cost-based optimizer in creating the best possible query plan, calculate statistics on this new table.

```
ANALYZE bb_structure_stations;
```

You created an Iceberg table with the ORC file format as you want this Structure layer table to be performance-oriented and transaction capable.

At this point in the pipeline, you would normally move the underlying data lake files from the `external_location` of the `bb_land_temp_stations` table to the data lake folder maintaining all the historical raw data received in the past. As this particular table's future ingestion files will be complete copies of the stations data, the data lake directory location should allow for creating a folder based on the date embedded in the raw file.

No table will be needed, but having the historical raw data allows flexibility should a prior version of the data be needed. You do not have S3 access to make changes, so you will not move the ingest file from its current location.

Step 8 - Simulate an ingestion of delta records

Now that the initial sync-up of the stations table is in the Structure layer, you can focus on building out the pipeline further. New ingest files for stations only include change records (deltas). These changes only address updates to existing stations and the creation of new stations. The output below shows a list of deltas for this table.

Note: To aid in readability and to avoid line-wrapping of these wide records, the "\" character is used to indicate the next line of indented text is still part of the record that started in the first column.

```
Last Updated,7/4/2022,,,,,,
Number,Name,Lat,Long,District,Public,Total docks,Deployment Year
NEW901,123 Main St,42.3625,-71.08822,District 12,No,22,2022
A32032,Airport MBTA Stop - Bremen St at Brooks St,42.37410288, \
     -71.03276432,Boston,Yes,11,2016
C32012,Andrew MBTA Stop - Dorchester Ave at Dexter St,42.33073333, \
     -71.05699851,Boston,Yes,0,2013
B32004,Aquarium MBTA Stop - 200 Atlantic Ave,42.35991176, \
     -71.05142981,Boston,No,15,2011
A32049,Bennington St at Another St,42.38522394,-71.01063069, \
     Boston,Yes,21,2022
```

Upon review from the original data, you determine there are two new stations and three updated stations.

As identified in **Step 4**, the pipeline's ingestion process is being simulated. To accomplish this, drop the temporary ingest table and recreate it with a different `external_location` value pointing to the incremental ingest data for July 4, 2022.


```
DROP TABLE bb_land_temp_stations;
CREATE TABLE bb_land_temp_stations (
  station_nbr varchar,
  name varchar,
  latitude decimal(8,6),
  longitude decimal(9,6),
  district varchar,
  public varchar,
  total_docks int,
  deployment_year varchar
) WITH (
 type = 'HIVE', format = 'TEXTFILE', textfile_field_separator = ',',
 skip_header_line_count = 2,
 external_location =
's3://edu-train-galaxy/bluebikes/raw_stations-2022-07-04_deltas/'
);
```

Verify that the five delta records are located in the `bb_land_temp_stations` table.

```
SELECT station_nbr, name, public, total_docks
  FROM bb_land_temp_stations;
```

Step 9 - Build the pipeline's `MERGE` statement

As you are dealing with a list of new and/or updated stations, you need to create a `MERGE` statement to apply the transformed delta records against.

The statement is broken into five parts. Let's step through them individually to understand how it works.

After you have reviewed this analysis, assemble & execute the entire query in the **Query editor**.

Part 1

This initial part identifies the Structure table. This is where the deltas will be merged.

```
-- merge into the Structure table
MERGE INTO bb_structure_stations
AS base
```

Part 2

The next part invokes the `USING` clause. This clause can be used to reference a simple table or the results of some other SQL. In this case, use the transformation query created in Step 6 as the cleaned set of delta records.

```
-- the contents from the using clause
USING (  -- transform the delta records
   SELECT
        station_nbr, name, latitude, longitude,
        nullif(district, '') AS district,
        starts_with(public, 'Yes') AS public,
        total_docks,
        try_cast(IF(deployment_year = 'N/A',
                    'NULL', deployment_year
                ) AS int
        ) AS deployment_year
   FROM bb_land_temp_stations
) AS deltas
```

Part 3

The next element of the query identifies the criteria necessary for a match between the base table and the list of deltas. For this scenario, this is a simple equality check on the `station_nbr` table.

```
-- look for a match on station_nbr
ON (base.station_nbr = deltas.station_nbr)
```

Part 4

The next section presents the actions that should follow a positive match. In this case, when a match is found, an `UPDATE` command is used to modify an existing record.

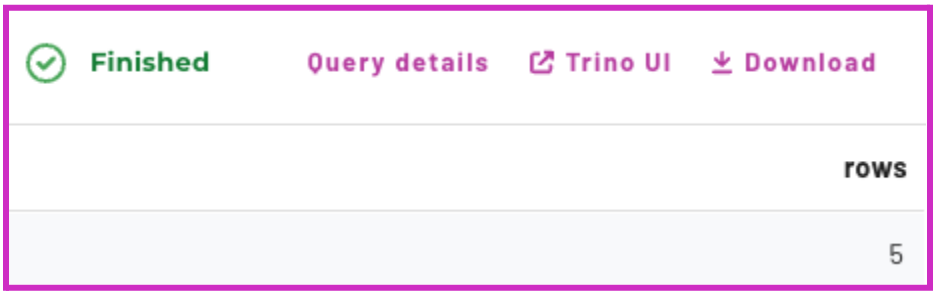

```
-- if a match then the records needs to be updated
WHEN MATCHED THEN
  UPDATE SET station_nbr = deltas.station_nbr,
             name = deltas.name,
             latitude = deltas.latitude,
             longitude = deltas.longitude,
             district = deltas.district,
             public = deltas.public,
             total_docks = deltas.total_docks,
             deployment_year = deltas.deployment_year
```

Part 5

Lastly, you need to account for the false condition and tell the query what to do in such cases. In this case, when a match is not found, then the delta record should be recorded in a new entry. The code below creates that entry.

```
-- if no match found then add the record
WHEN NOT MATCHED THEN
  INSERT VALUES (deltas.station_nbr,  deltas.name,
    deltas.latitude, deltas.longitude, deltas.district,
    deltas.public, deltas.total_docks,
    deltas.deployment_year);
```

Now that you've understood how **PART 1** - **PART 5** work, it's time to execute them. Combine the SQL from each part into one large query and execute it in your Query editor. You will see a notification similar to the image below, indicating that 5 rows have been changed.



Step 10 - Verify the `MERGE` logic

Since the Iceberg table format maintains prior snapshots that are queryable via time travel, we can use those as reference points corresponding to each snapshot ID. Using this approach, you will be able to compare the current state of the table with the prior state held in the snapshot.

```
SELECT * FROM "bb_structure_stations$snapshots";
```

committed_at	snapshot_id	parent_id	operation
2023-03-02 21:59:07.8...	1327769257252751 58	NULL	append
2023-03-02 23:26:14.0...	2034244356527960003	1327769257252751 58	overwrite

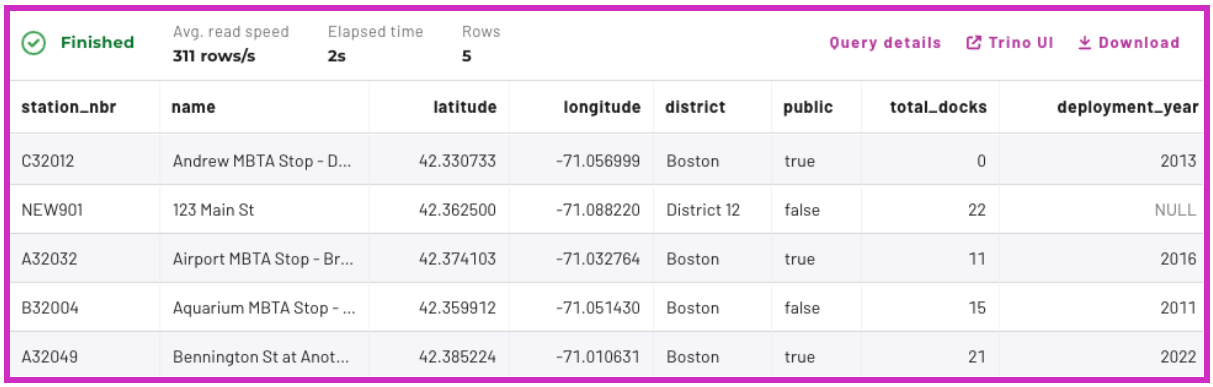
The value of the `snapshot_id` listed in the first row is the initial snapshot ID.

Using the code below, replace `999999999` with the value that your query returned for the original snapshot.

Note: The value listed in your table will be different from the one listed by the instructor's table.

```
SELECT * FROM bb_structure_stations
EXCEPT
SELECT * FROM bb_structure_stations
FOR VERSION AS OF 999999999;
```

You should see output similar to the image below, listing the 5 rows from the delta ingestion file.



Finished | Avg. read speed 311 rows/s | Elapsed time 2s | Rows 5 | Query details | Trino UI | Download

station_nbr	name	latitude	longitude	district	public	total_docks	deployment_year
C32012	Andrew MBTA Stop - D...	42.330733	-71.056999	Boston	true	0	2013
NEW901	123 Main St	42.362500	-71.088220	District 12	false	22	NULL
A32032	Airport MBTA Stop - Br...	42.374103	-71.032764	Boston	true	11	2016
B32004	Aquarium MBTA Stop - ...	42.359912	-71.051430	Boston	false	15	2011
A32049	Bennington St at Anot...	42.385224	-71.010631	Boston	true	21	2022

Step 11 - Next steps

Typically, the next step would involve moving the temporary ingestion table's underlying file to the location of the raw table with all historical information. In this demo environment, you cannot perform this step as you do not have access directly to S3.

Often, in production, the next step might also involve the automation of this pipeline. This would shift away from manual execution and help save time.

END OF LAB EXERCISE