Implementing the medallion architecture with Starburst

Lab Guide (v1.0.0)

Table of Contents

Getting started

Lab 1: Create student account (5 mins)

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.

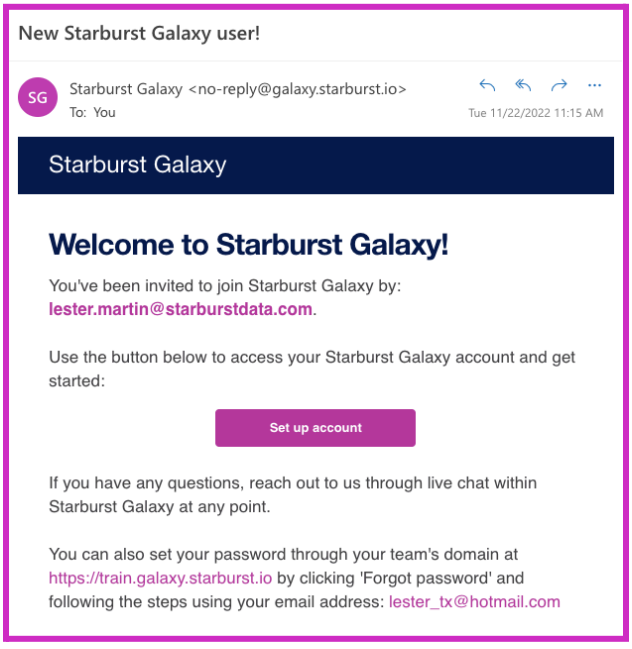
Activities

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

Step 1 - Set up account using invitation email

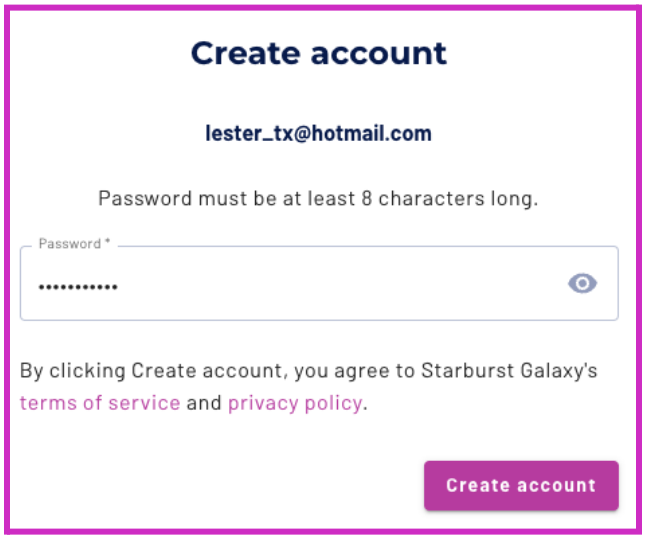
Your instructor has sent you an invitation to create a student account. The system will send an email similar to the one pictured below.

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



Step 2 - Create password

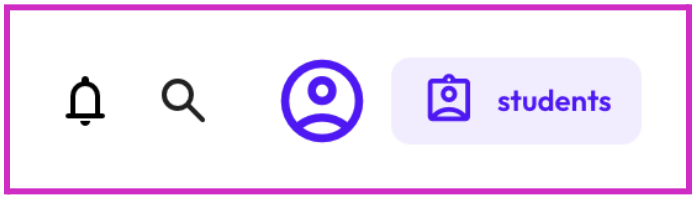
Next, you must create an account password. Enter a password that meets the minimum requirements, then click the **Create account** button.



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

Lab 2: Execute queries in Starburst Galaxy (15 mins)

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

- Getting started - Lab 1: Create student account

Activities

1. Sign in and verify role
2. Confirm cluster is running
3. Navigate the cluster list
4. Select `sf1` schema from `tpch table`
5. Access shared tables and execute `SELECT` query
6. Execute `DESCRIBE` query
7. Run `SELECT COUNT (*)` and `SELECT *` queries
8. Downloading results
9. Explore as you see fit
10. Additional UI features

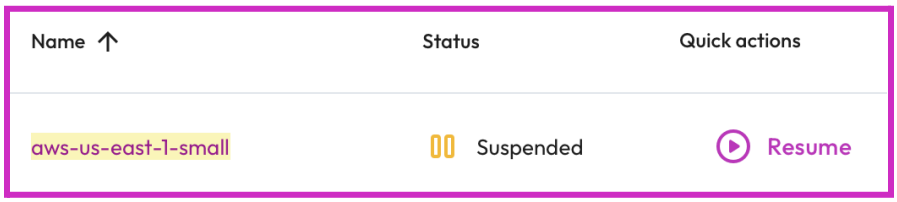
Step 1 - Sign in and verify role

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

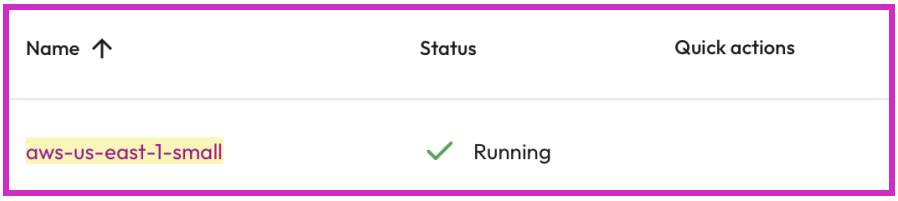
Step 2 - Confirm cluster is running

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

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

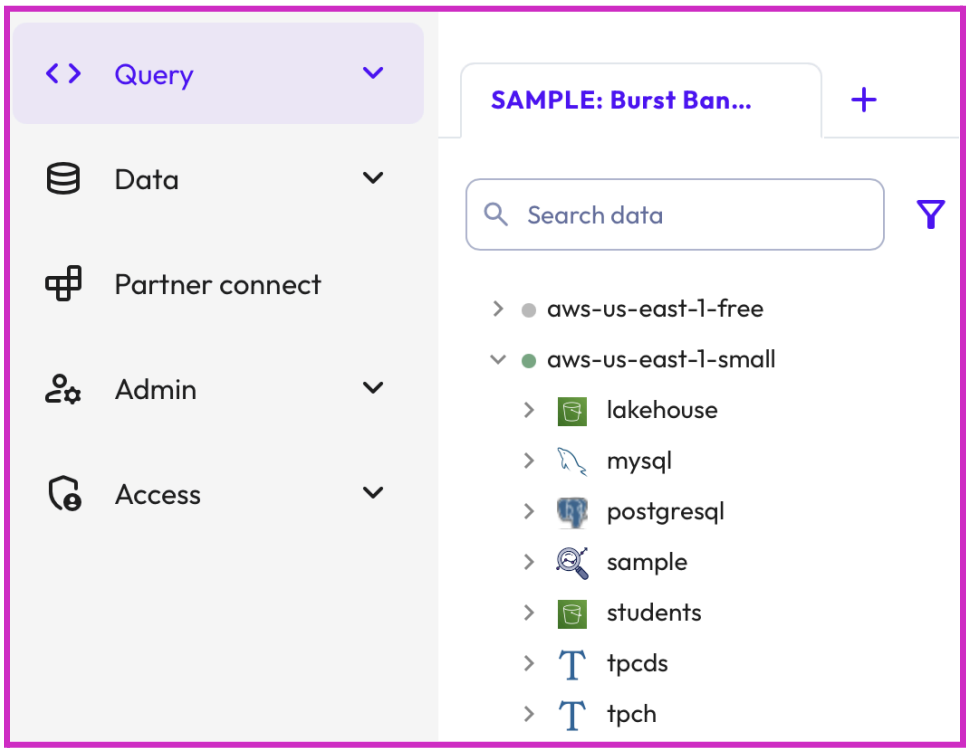


To restart the cluster, select **Resume** (or **Start**) from the **Quick actions** column. Confirm that the cluster is reporting a **Status** of `Running` before continuing, similar to the image below.



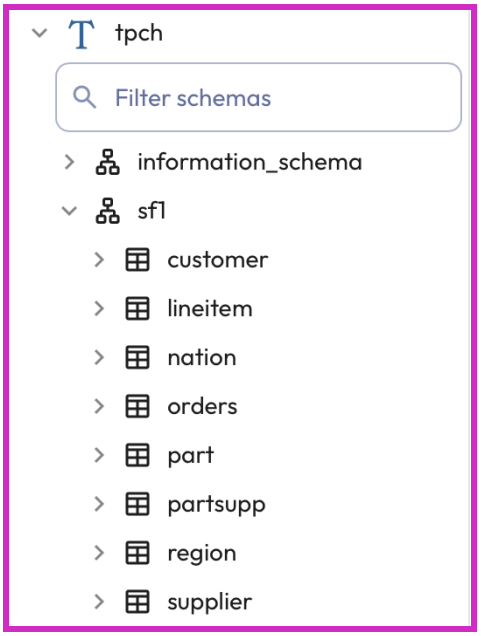
Step 3 - Navigate the cluster list

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



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

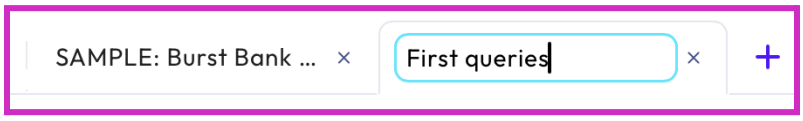
From the list of catalogs presented, expand `tpch` and then expand the `sf1` schema to see a list of tables.



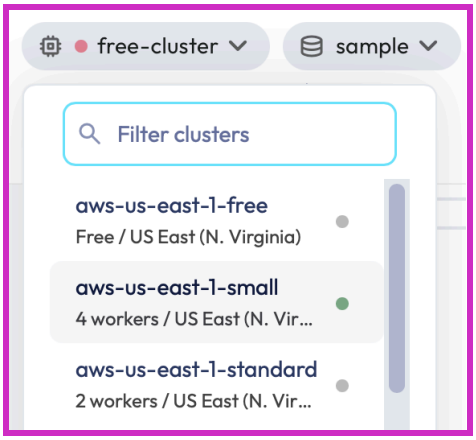
Step 5 - Access shared tables and execute `SELECT` query

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.

Click on the default name of the tab which will be a timestamp of when you created it. Change the name to something more memorable



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-small` from the drop-down menu.

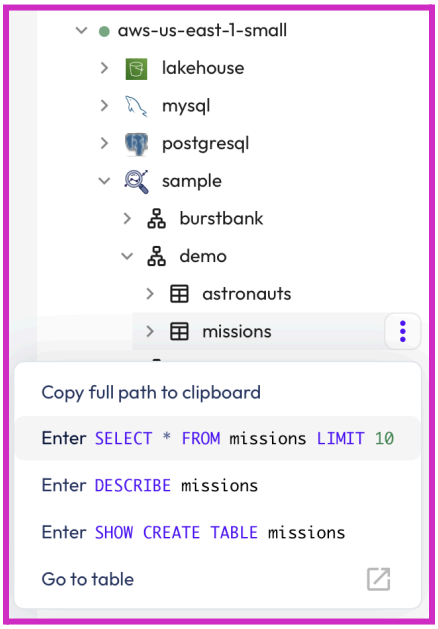


You now need to select the correct catalog, schema, and table before running the query.

Use the cluster list at the left of the editor pane and perform the following actions starting from the expanded view of the `aws-us-east-1-small` cluster:

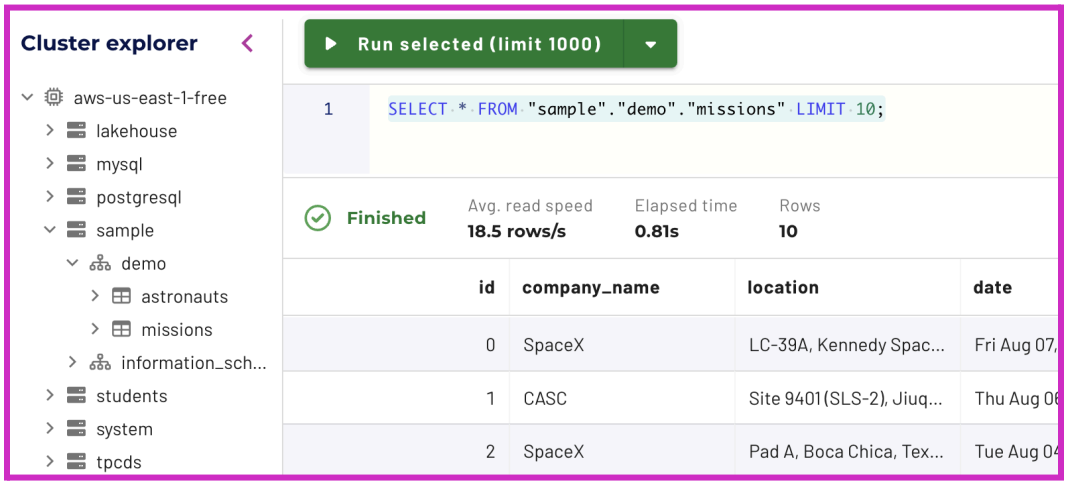
> **Note**: *The cluster should already be running. If it is not, it will pause for a short period before it can be used. This behavior is entirely normal.*

- 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.



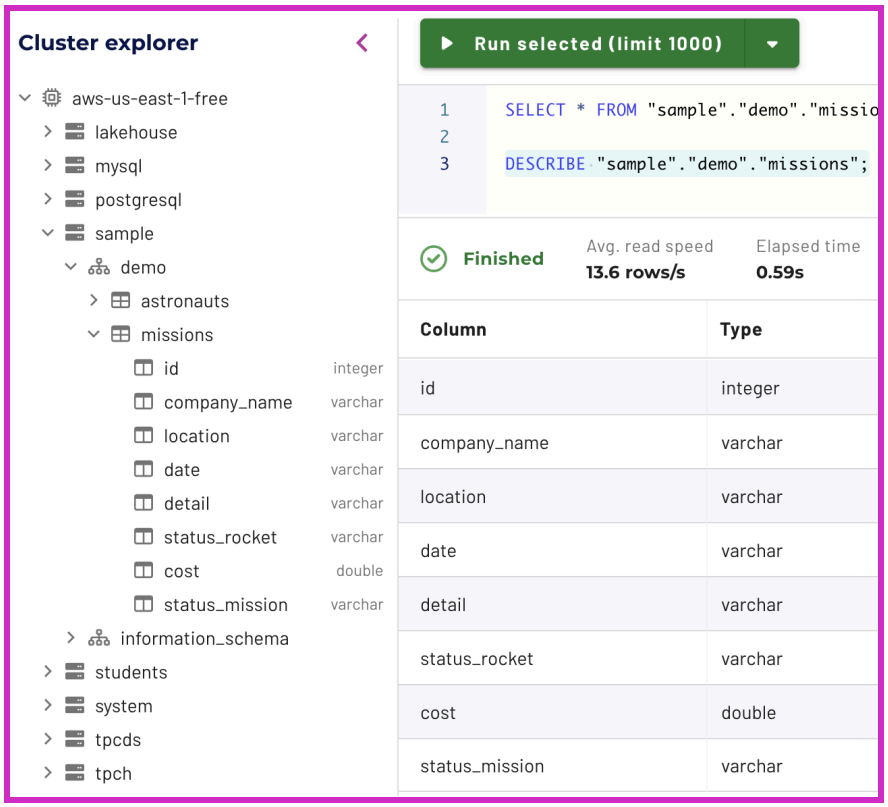
The query should now be copied into the query editor.

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 6 - 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 7 - Run `SELECT COUNT(*)` and `SELECT *` queries

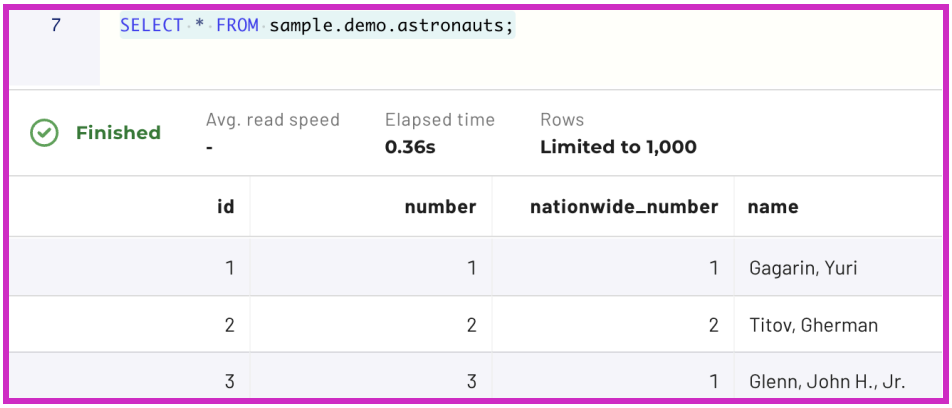
In the query editor, run a query on the `astronauts` table to verify there are 1279 rows present.

```
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?



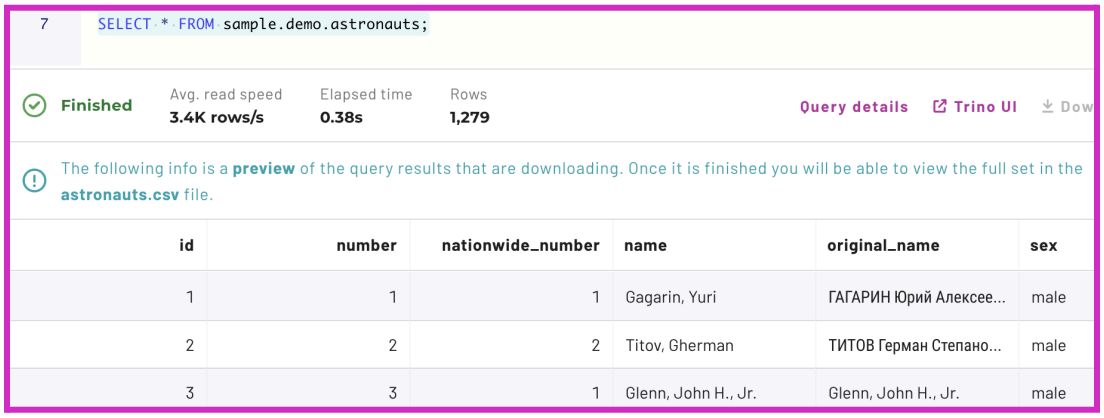
```
7  SELECT * FROM sample.demo.astronauts;
```

Finished | Avg. read speed: - | Elapsed time: 0.36s | Rows: Limited to 1,000

id	number	nationwide_number	name
1	1	1	Gagarin, Yuri
2	2	2	Titov, Gherman
3	3	1	Glenn, John H., Jr.

Step 8 - 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?



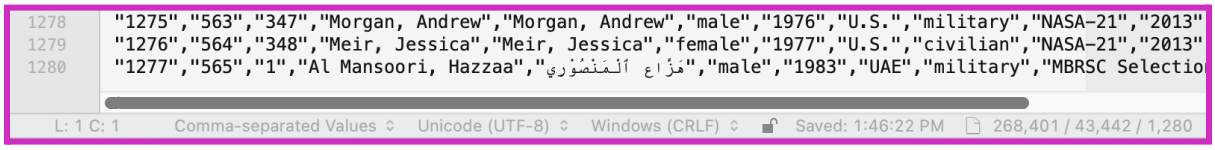
```
7  SELECT * FROM sample.demo.astronauts;
```

Finished | Avg. read speed: 3.4K rows/s | Elapsed time: 0.38s | Rows: 1,279 | Query details | Trino UI | Dow

The following info is a **preview** of the query results that are downloading. Once it is finished you will be able to view the full set in the **astronauts.csv** file.

id	number	nationwide_number	name	original_name	sex
1	1	1	Gagarin, Yuri	ГАГАРИН Юрий Алексее...	male
2	2	2	Titov, Gherman	ТИТОВ Герман Степано...	male
3	3	1	Glenn, John H., Jr.	Glenn, John H., Jr.	male

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



```
1278  "1275","563","347","Morgan, Andrew","Morgan, Andrew","male","1976","U.S.","military","NASA-21","2013"
1279  "1276","564","348","Meir, Jessica","Meir, Jessica","female","1977","U.S.","civilian","NASA-21","2013"
1280  "1277","565","1","Al Mansoori, Hazzaa","هَزَّاع ٱلْمَنْصُوْرِي","male","1983","UAE","military","MBRSC Selectio
```

L: 1 C: 1 | Comma-separated Values | Unicode (UTF-8) | Windows (CRLF) | Saved: 1:46:22 PM | 268,401 / 43,442 / 1,280

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. *While not part of these exercises,* inserting the results into a new or existing table defined to use the file format you desire will give you lots of flexibility in this situation.

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

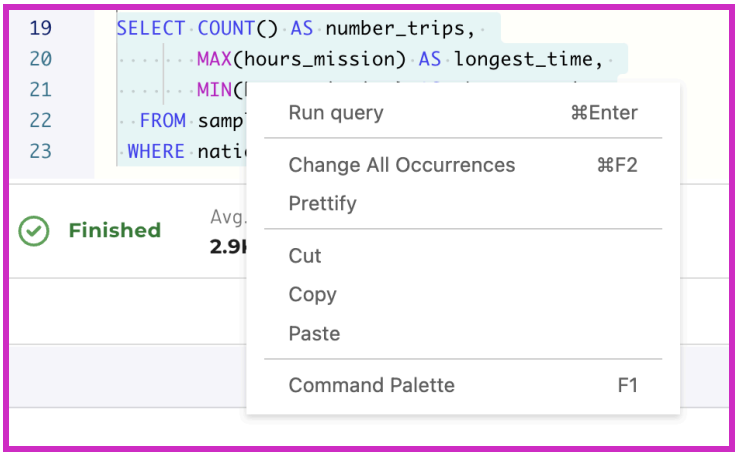
```
DESCRIBE sample.demo.astronauts;
```

Step 9 - Explore as you see fit

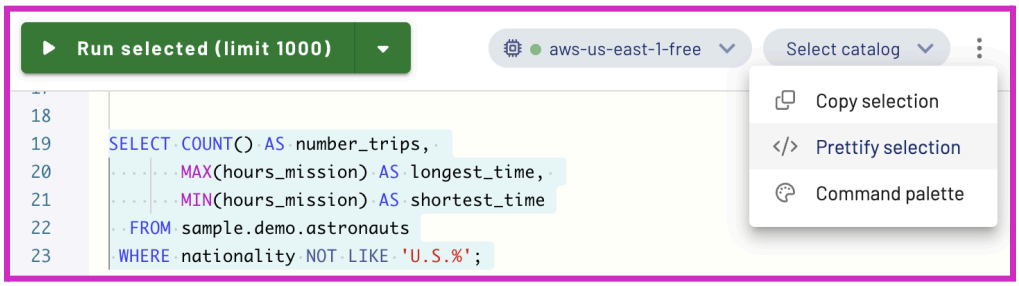
Optionally, run any query on this table as you would like. Here are a few examples.

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

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

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

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

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.

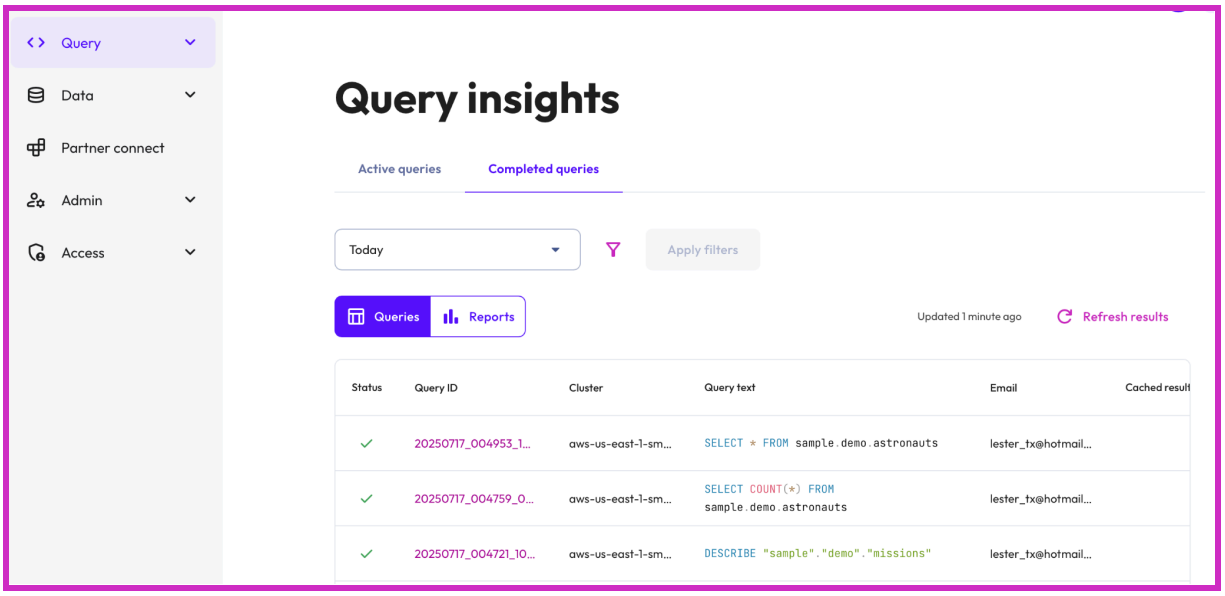


Regardless of which way you exercise this feature, the last example query should now look similar to this.

```
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.%';
```

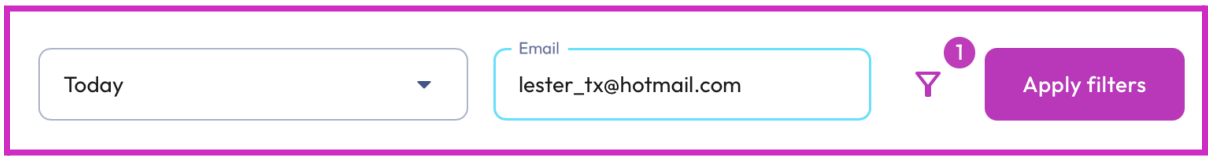
Step 10 - Additional UI features

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



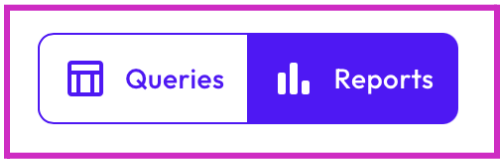
This shows queries across all users. Click the funnel icon shown below.

In the pop-up menu that surfaces select **Email** and fill in your email address in the box that surfaces before clicking the **Apply filters** button (below) to limit the results to your queries.



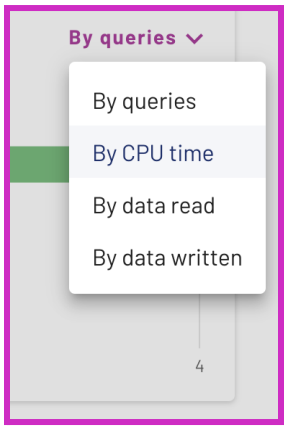
Feel free to experiment with additional filters. Share anything you find interesting with the class or the instructor. When done, click **Reset filters** to see all **Queries**.

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

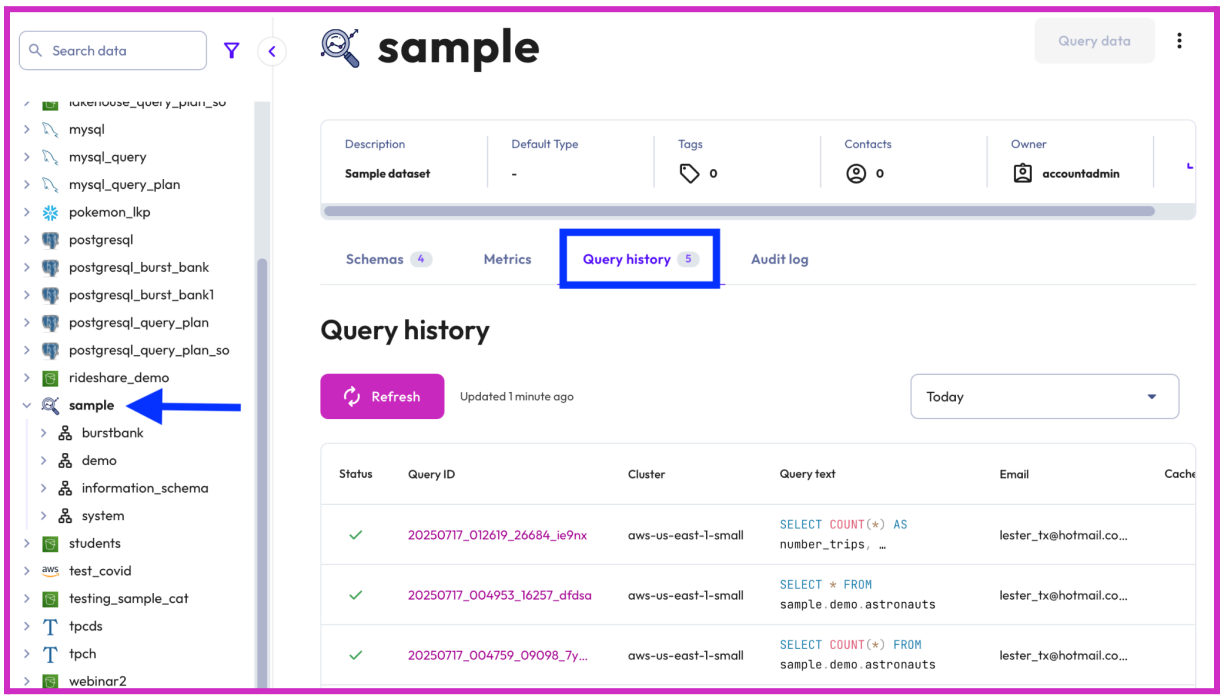


What is the highest number of **Queries over time**? What times were the peaks?

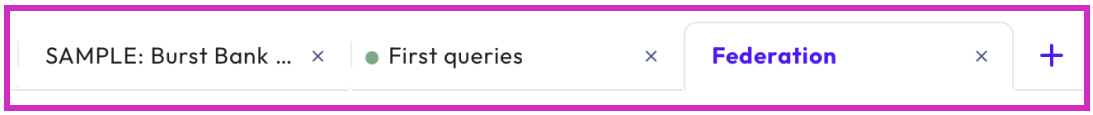
Review the **Top users** report to determine who has run the most queries. How many queries did this user run? If you were not the top user, how many queries did you run? Did the top user change when you toggled **By queries** to **By CPU time** on the right side of this report?



Navigate to **Data** > **Catalogs** from the left navigation. Expand `sample` 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** > **Saved queries**, and just below your list of Recent queries select **Shared with you** tab. Open the `Federation` query in the list.



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

END OF LAB EXERCISE

Lab 3: Create and populate Iceberg tables (10 mins)

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 the Parquet file types, add records, and query metadata values using Iceberg.

Prerequisites

- Getting started - Lab 1: Create student account

Activities

1. Prepare for the exercise
2. Create Iceberg tables using Parquet
3. Research Iceberg connector documentation
4. Add records to Iceberg table
5. Simplify file list results
6. Find distinct values
7. Review documentation and list files
8. Add additional records to 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-small` **Cluster** is reporting a **Status** of `Running`.
- In the **Query editor**, select `aws-us-east-1-small` in the cluster pulldown.

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;
```

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 - 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 5 - 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 6 - 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 7 - Review documentation and list files

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 6**. 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 8 - 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 7** 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

OPTIONAL Lab: Data modifications and snapshots with Iceberg (20 mins)

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

- Getting started - 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-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 - 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`).

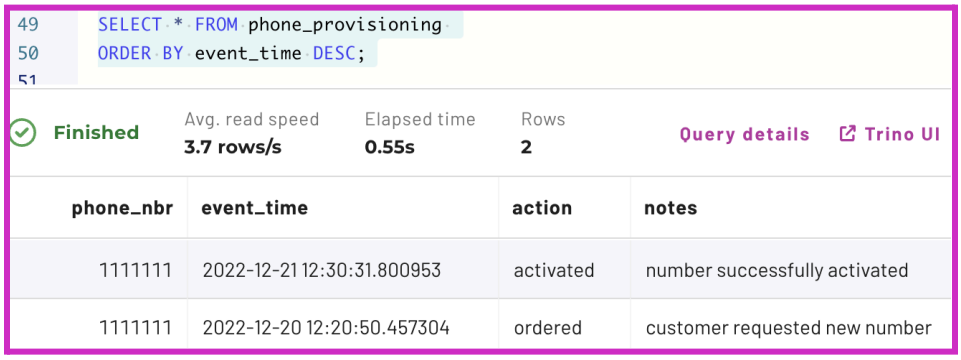
```
SELECT * FROM phone_provisioning
EXCEPT
SELECT * FROM phone_provisioning
FOR VERSION AS OF 1234567890;
```

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;
```



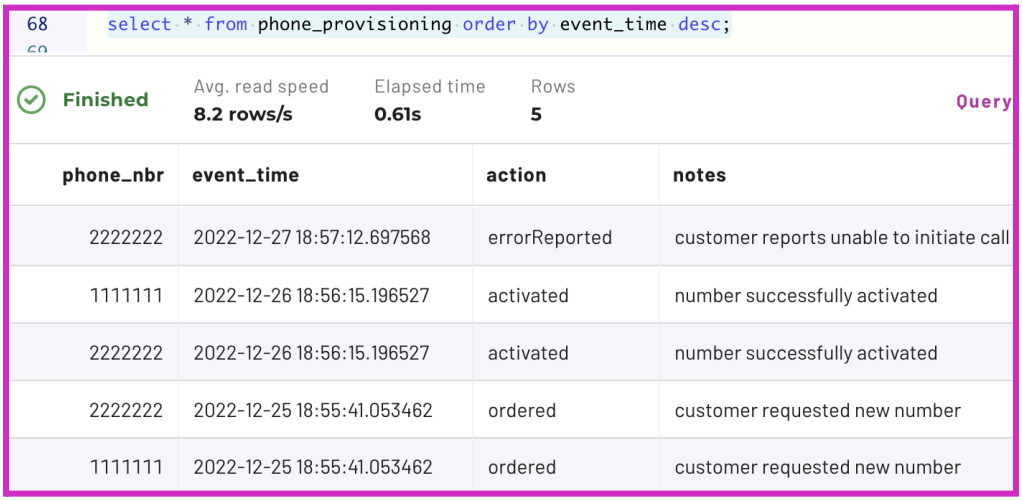
It was determined that the `DELETE` was a mistake and needs to be recovered from. Use the `snapshot_id` you saved earlier in the following command that rolls the table back to before the deletions.

```
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

OPTIONAL Lab: Exercise advanced features of Iceberg (15 mins)

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

- Getting started - 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-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 - Create table

In this lab, you will build and populate a table with airplane data.

Using the `aws-us-east-1-small` 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;
```

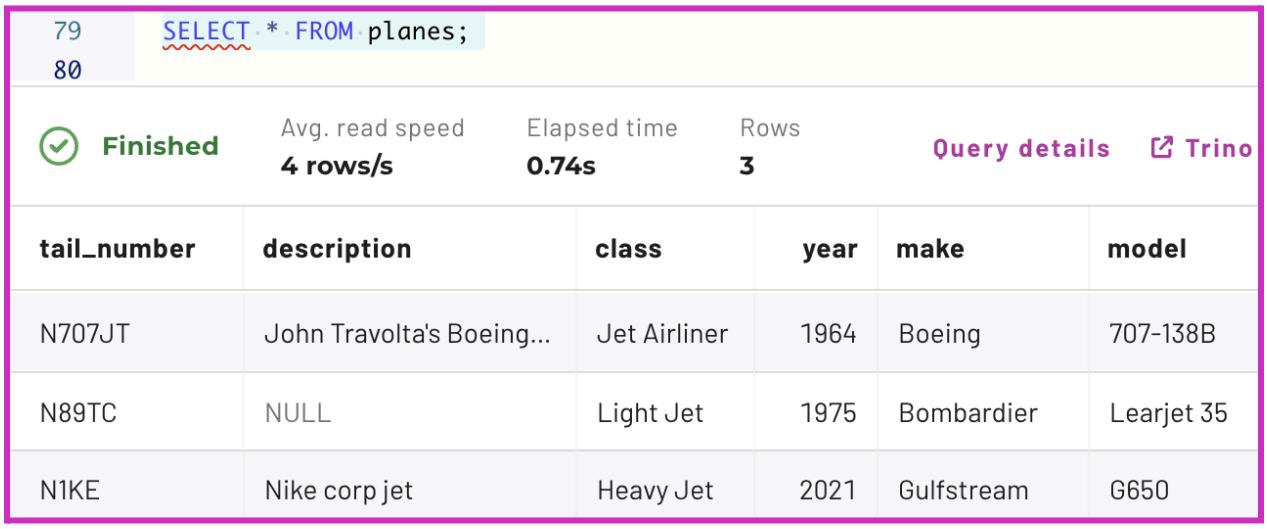
Step 7 - Modify existing records using new column layout

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'];
```

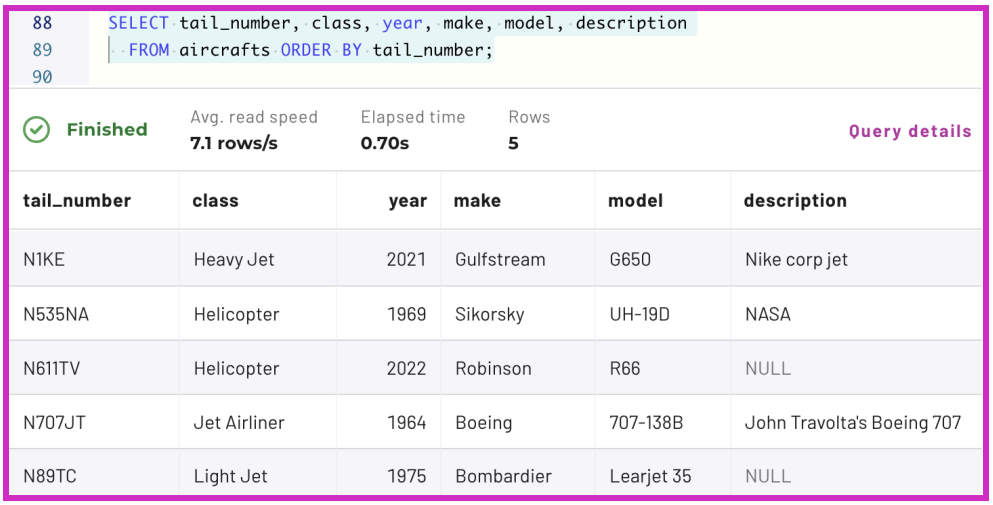
Step 9 - Add two new helicopters into modified table

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[];
```

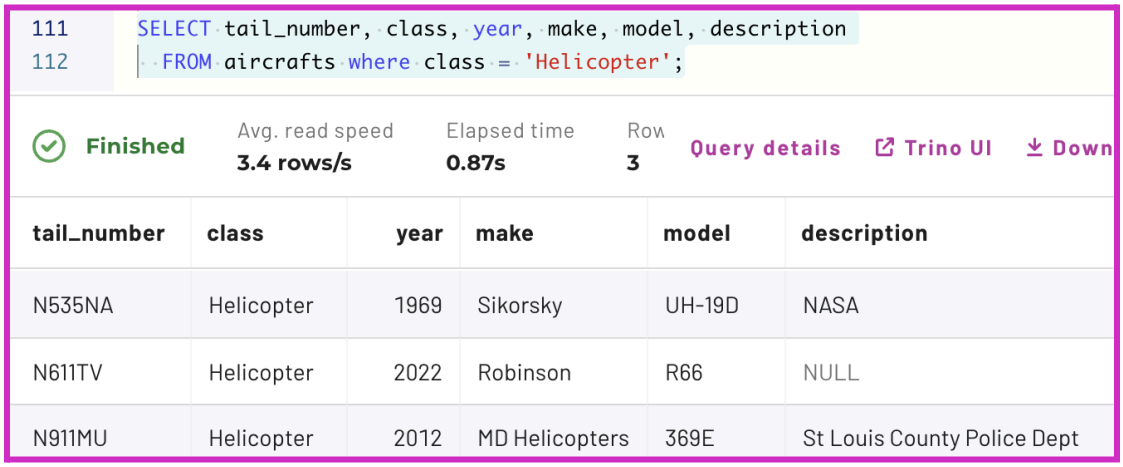
Step 12 - Add new record to table after partitions eliminated

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';
```



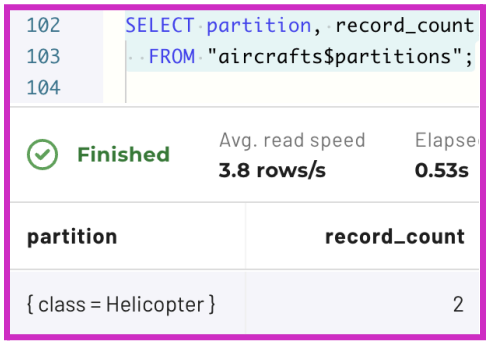
```
111  SELECT tail_number, class, year, make, model, description
112    FROM aircrafts where class = 'Helicopter';
```

Finished | Avg. read speed 3.4 rows/s | Elapsed time 0.87s | Row 3 | Query details | Trino UI | Down

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";
```



```
102  SELECT partition, record_count
103    FROM "aircrafts$partitions";
104
```

Finished | Avg. read speed 3.8 rows/s | Elapse 0.53s

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

Data pipelines & data products

Lab 1: Construct a pipeline with insert-only transactions (45 mins)

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

- Getting started - 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

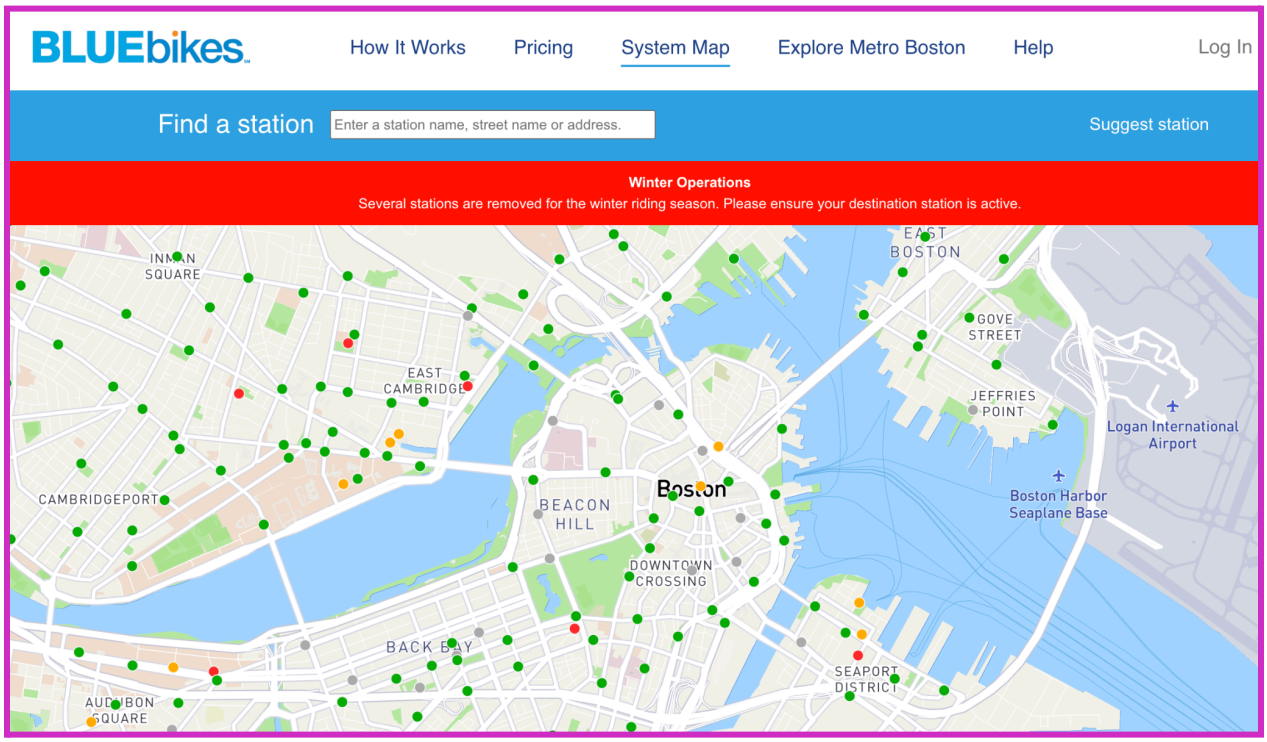
- Sign in and verify the `students` role is selected in the upper-right corner.
- Ensure the `aws-us-east-1-small` **Cluster** is reporting a **Status** of `Running`.
- In the **Query editor**, select `aws-us-east-1-small` in the cluster pulldown.

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 - 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 (Bronze) 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 (Silver) 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.

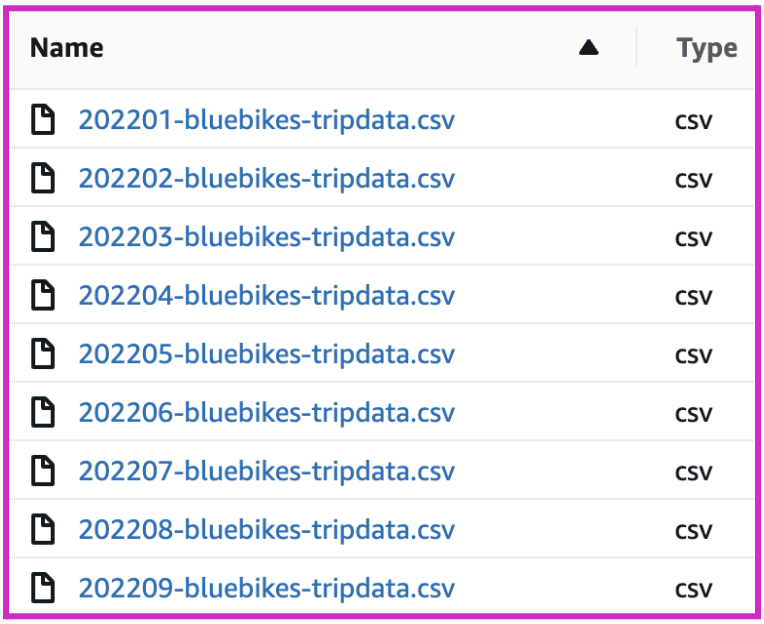
Consume (Gold) layer

- 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.

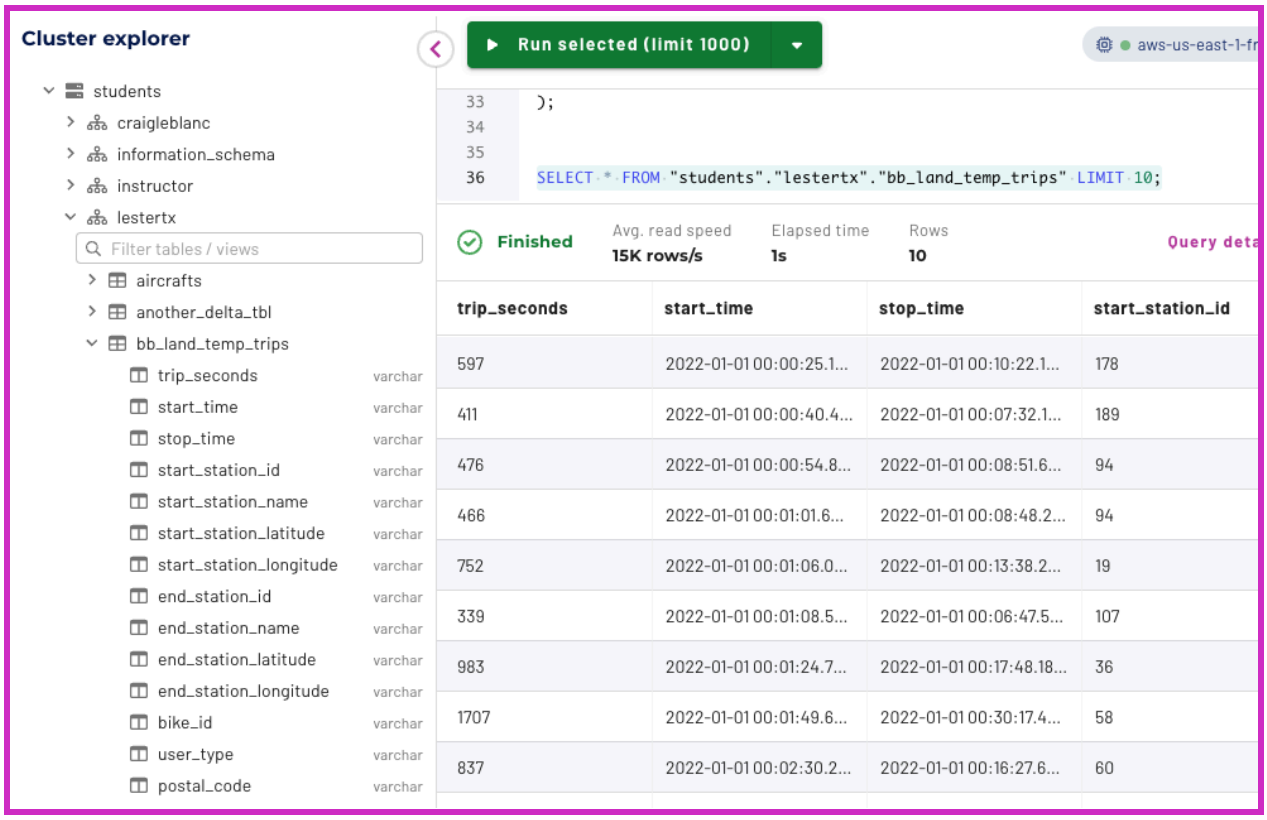
Create an ingest table in our logical Land layer to temporarily house the data being ingested.

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.

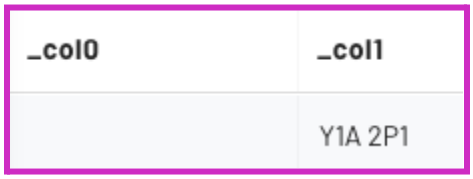
```
SELECT
   CAST(trip_seconds AS int) AS trip_seconds,
   CAST(start_time AS timestamp(6)) AS start_time,
   CAST(stop_time AS timestamp(6)) AS stop_time
FROM bb_land_temp_trips;
```

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.

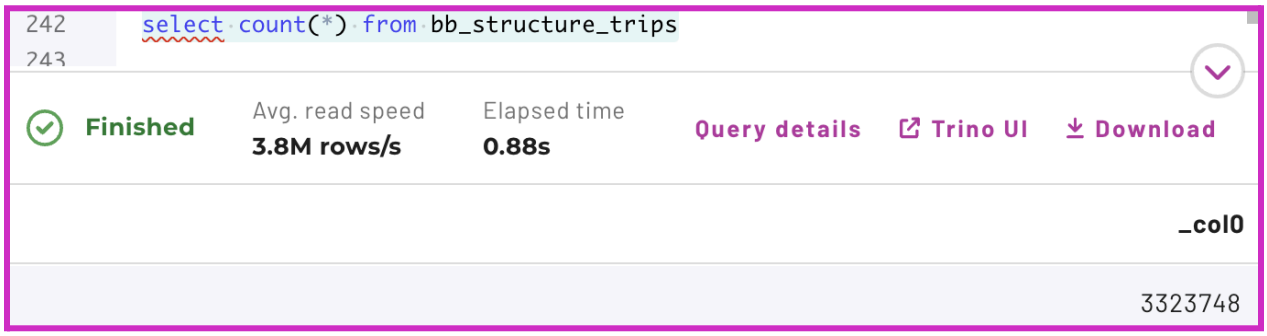
```
DROP TABLE bb_land_temp_trips;
CREATE TABLE bb_land_temp_trips (
trip_seconds varchar,
start_time varchar,
stop_time varchar,
start_station_id varchar,
start_station_name varchar,
start_station_latitude varchar,
start_station_longitude varchar,
end_station_id varchar,
end_station_name varchar,
end_station_latitude 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

OPTIONAL Lab : Construct a pipeline with the `MERGE` statement (20 mins)

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

- Getting started - Lab 1: Create student account

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-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 - 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.

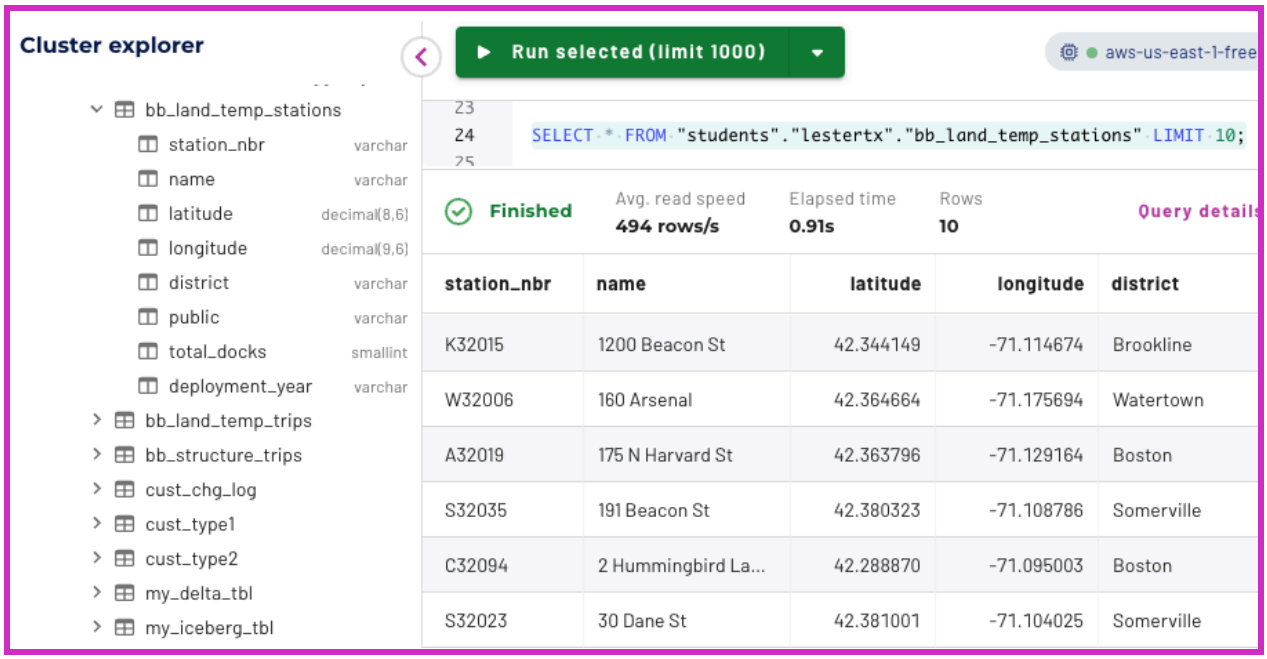
Create an ingest table in our logical Land layer to temporarily house the data being ingested.

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 from **Part 1** - **Part 5** below, 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.

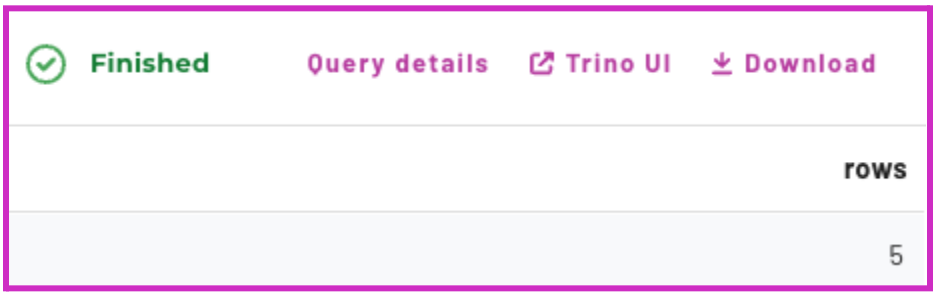
```
-- if a match then the records needs to be updated
WHEN MATCHED THEN
  UPDATE SET station_nbr = deltas.station_nbr,
             name = deltas.name,
             latitude = deltas.latitude,
             longitude = deltas.longitude,
             district = deltas.district,
             public = deltas.public,
             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 understand 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...	132776925725275158	NULL	append
2023-03-02 23:26:14.0...	2034244356527960003	132776925725275158	overwrite

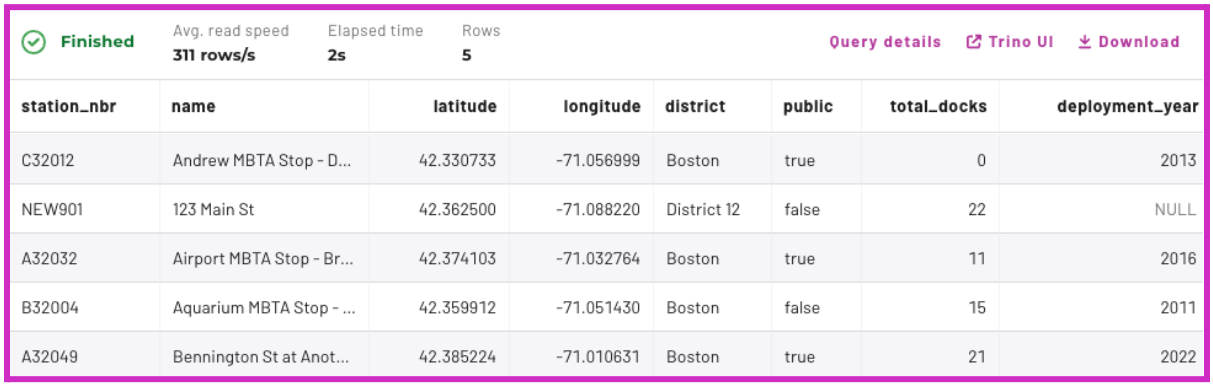
The value of the `snapshot_id` listed in the first row is the initial snapshot ID.

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

Lab 2: Produce and consume a data product (15 mins)

Learning objectives

- In this lab, you will learn how to promote an existing schema into a data product. Special focus will be placed on enhancing metadata to aid in human and/or AI based searches.

Prerequisites

- Getting started - Lab 1: Create student account
- Data pipelines & data products - Lab1: Construct a pipeline with insert-only transactions

Activities

1. Prepare for the exercise
2. Enhance your schema's metadata
3. Enhance table & column metadata
4. Promote schema to a data product
5. Enhance the data product
6. Consume the data product

Step 1 - Prepare for the exercise

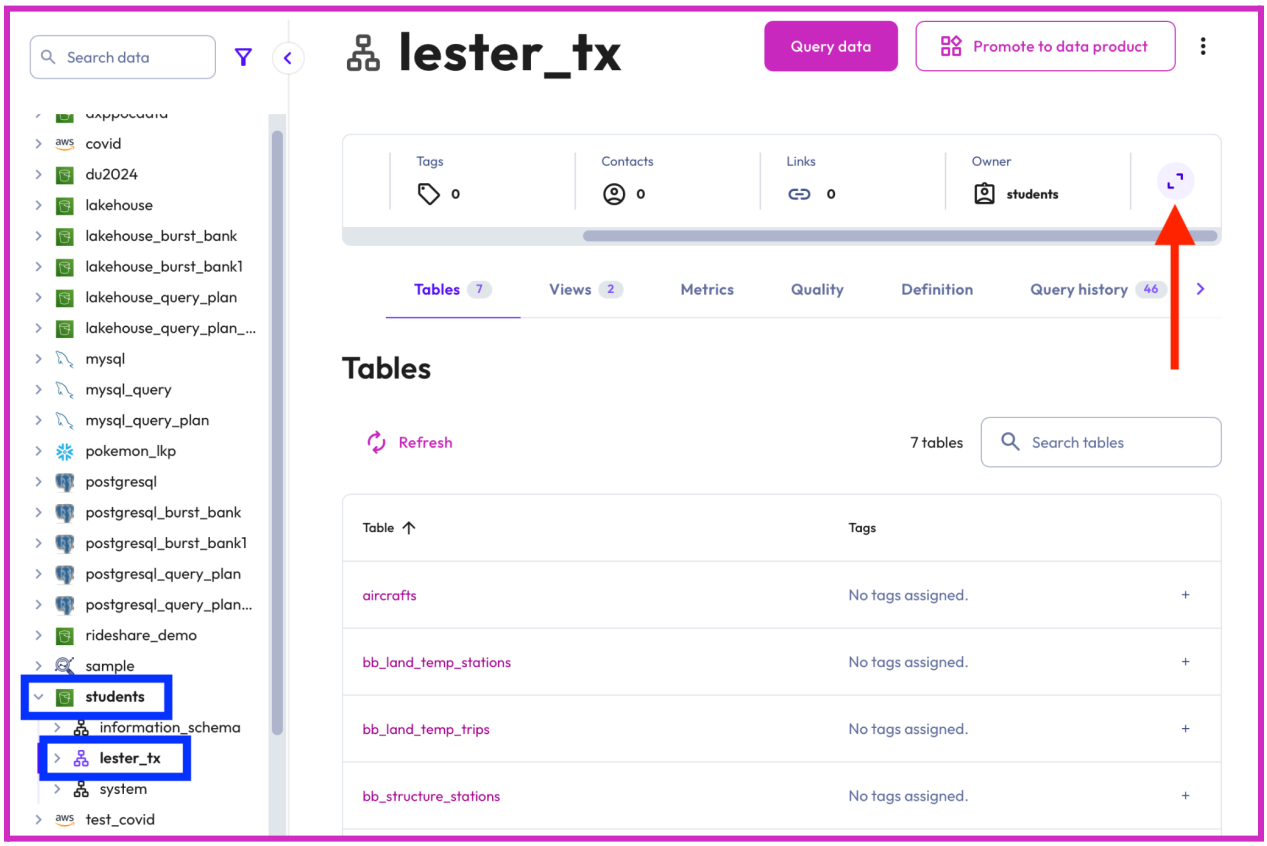
- Sign in and verify the `students` role is selected in the upper-right corner.
- Ensure the `aws-us-east-1-small` **Cluster** is reporting a **Status** of `Running`.
- In the **Query editor**, select `aws-us-east-1-small` in the cluster pulldown.

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.**

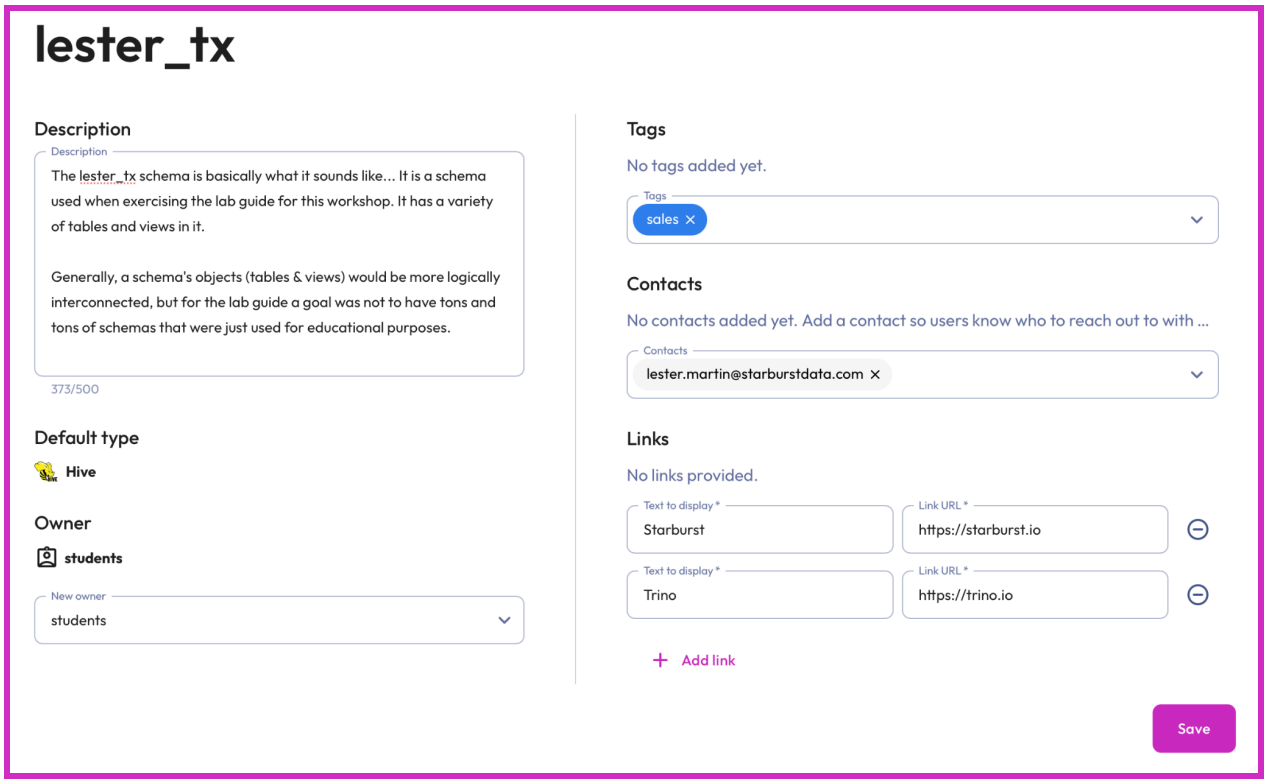
```
CREATE SCHEMA students.yourname;
```

Step 2 - Enhance your schema's metadata

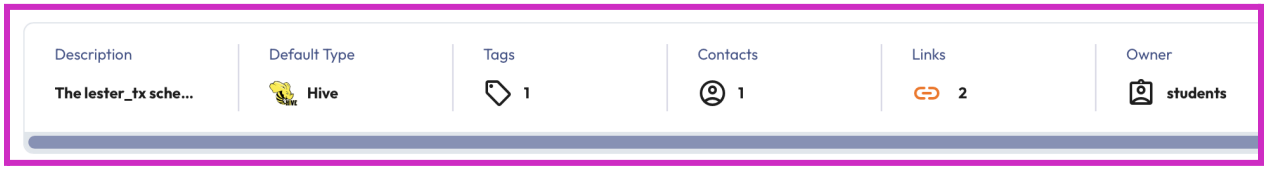
Navigate to **Data** > **Catalogs** from the left-hand menu. In the presented list of **Catalogs**, toggle open the **customers** catalog and then click on YOUR schema.



Click the **expander icon** identified by the red arrow in the prior image. In the schema-level metadata pop-up window that surfaces, add a **Description**, select **sales** from the **Tags** pulldown, identify yourself in the **Contacts**, and add 2 **Links**. Press **Save** to persist the changes to the schema metadata.

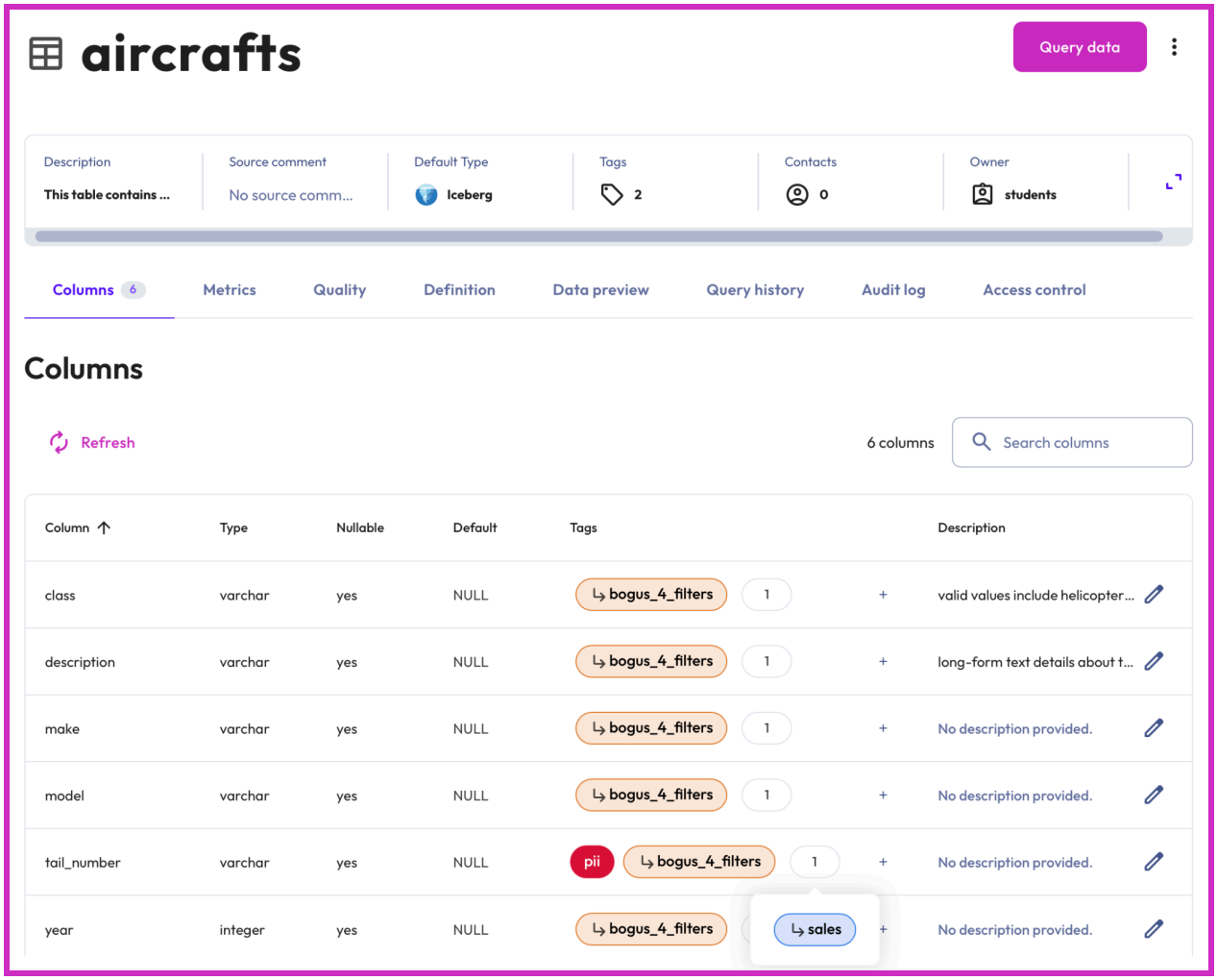


Notice that the information bar has more context now than before.



Step 3 - Enhance table & column metadata

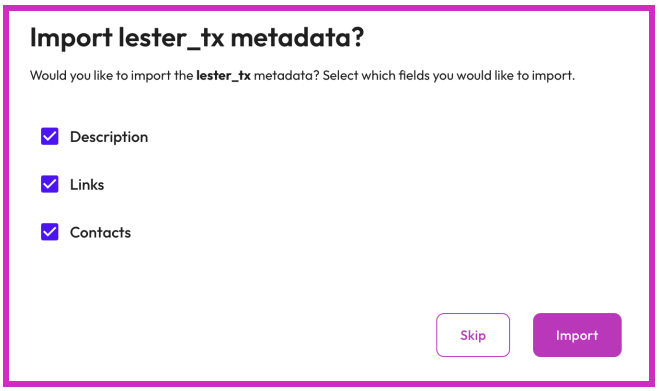
In the list of **Tables**, click on one of the names to see another information bar and a list of **Columns**. Click on the **expander icon** to create table-level metadata. Enhance the **Tags** and **Description** fields on a few **Columns**.



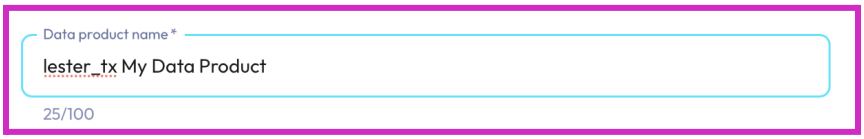
Step 4 - Promote schema to a data product

Return to YOUR schema as instructed in **Step 1**. Click on the **Promote to data product** button.

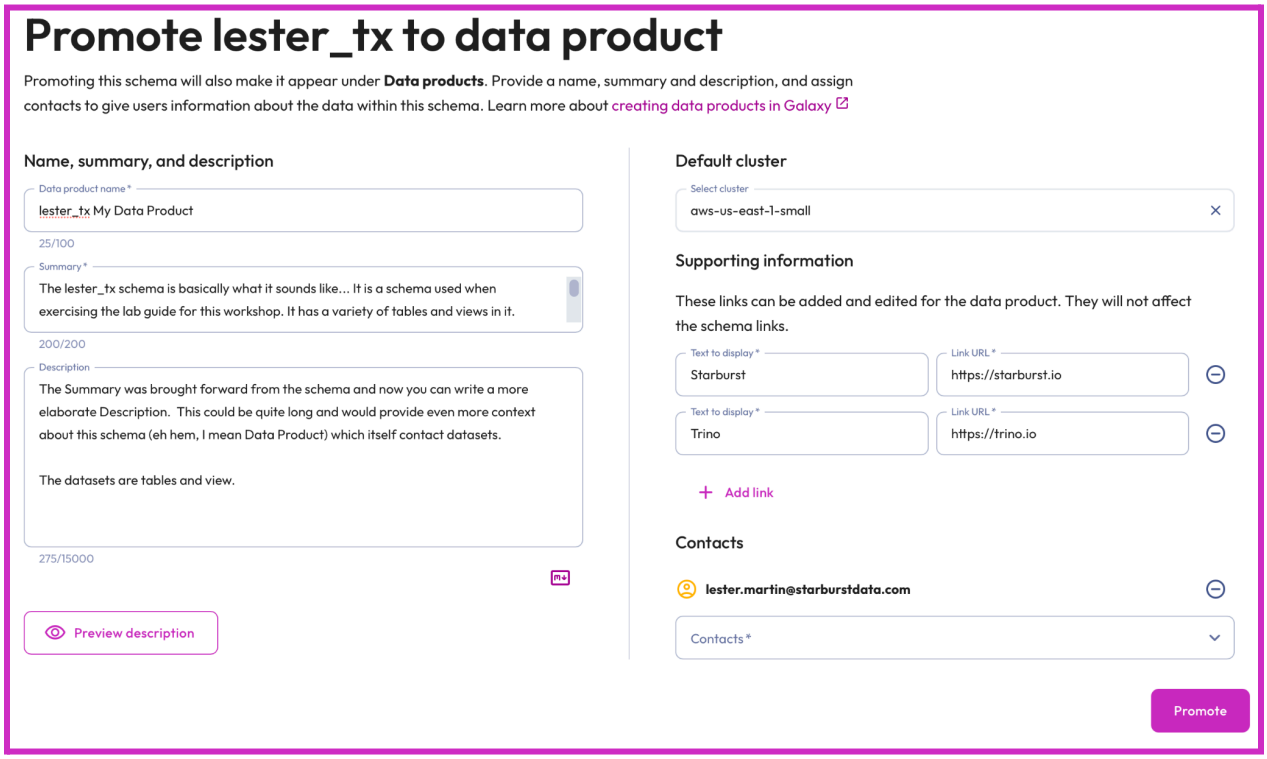
Keep the **import metadata** options selected and click on **Import**.



Prefix the Data product name with YOUR schema name.

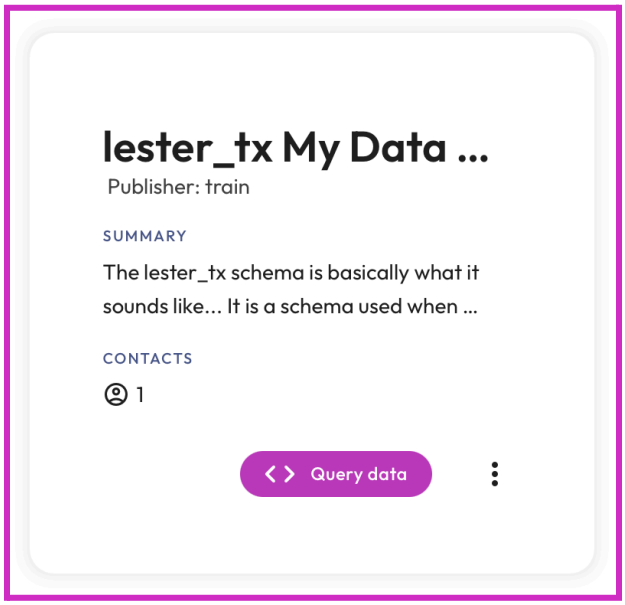


Provide some text for the **Description** field, select **aws-us-east-1-small** as the **Default cluster**, and click on the **Promote** button.

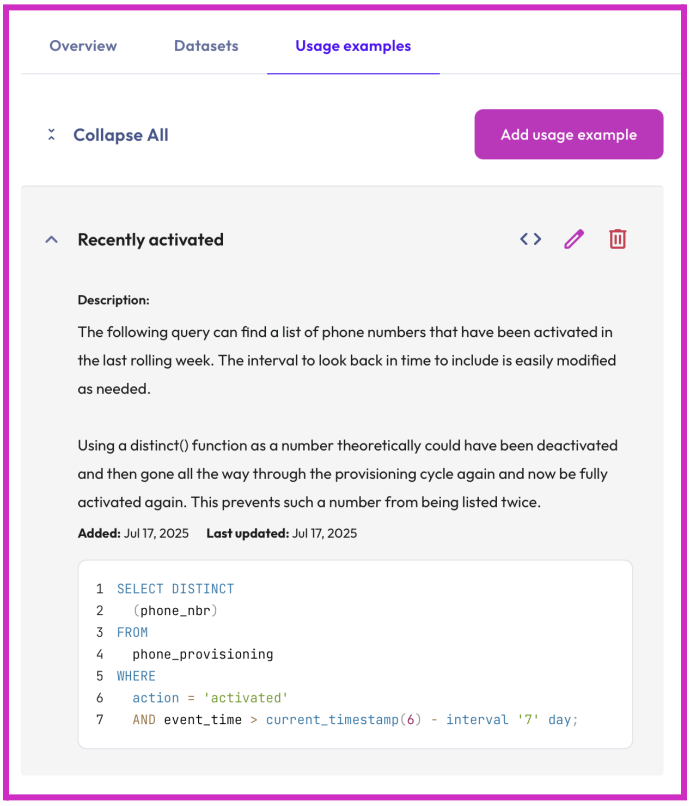


Step 5 - Enhance the data product

Click on the name of YOUR data product to view it.

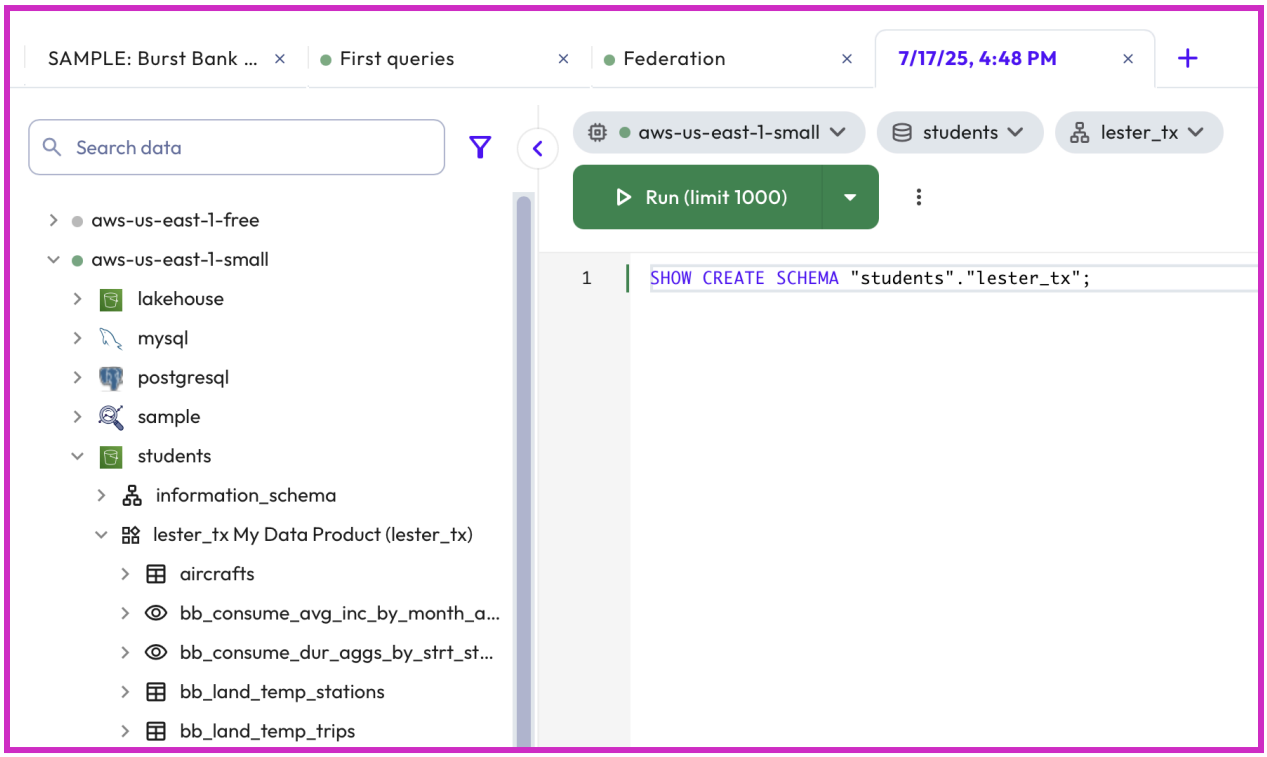


Explore the data product. Verify your previous metadata additions are present. Optionally, enhance the metadata about your data product further such as creating a **Usage example**.



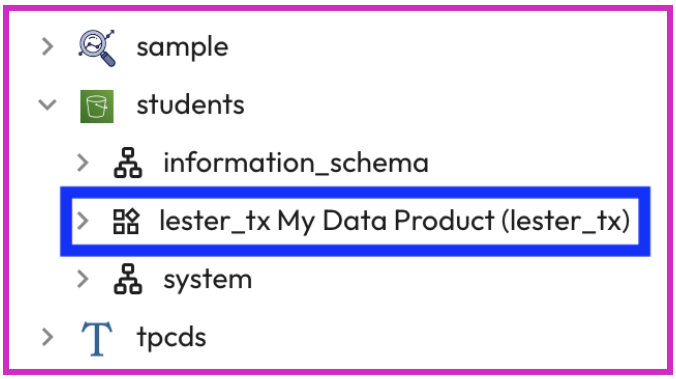
Step 3 - Consume the data product

Click on the **<> Query data** button to the right of the data product name. This will redirect you to a new tab in the **Query editor** and have the catalog and schema pulldowns preselected with the underlying schema the data product was promoted from.



From a data access & governance perspective, the data product is accessed by the actual schema name and using all the existing infrastructure and core query engine functionality.

Notice that the UI creates a special icon for a schema that is represented by a data product and includes both of their names.



END OF LAB EXERCISE

Managed Iceberg pipelines

Lab 1: Explore the file ingestion service (10 mins)

Learning objectives

- In this lab, you will configure and execute the file ingestion service.

Prerequisites

- Getting started - Lab 1: Create student account

Activities

1. Prepare for the exercise
2. Locate the interactive demo instructions
3. Perform demo steps
4. Stop live tables

Step 1 - Prepare for the exercise

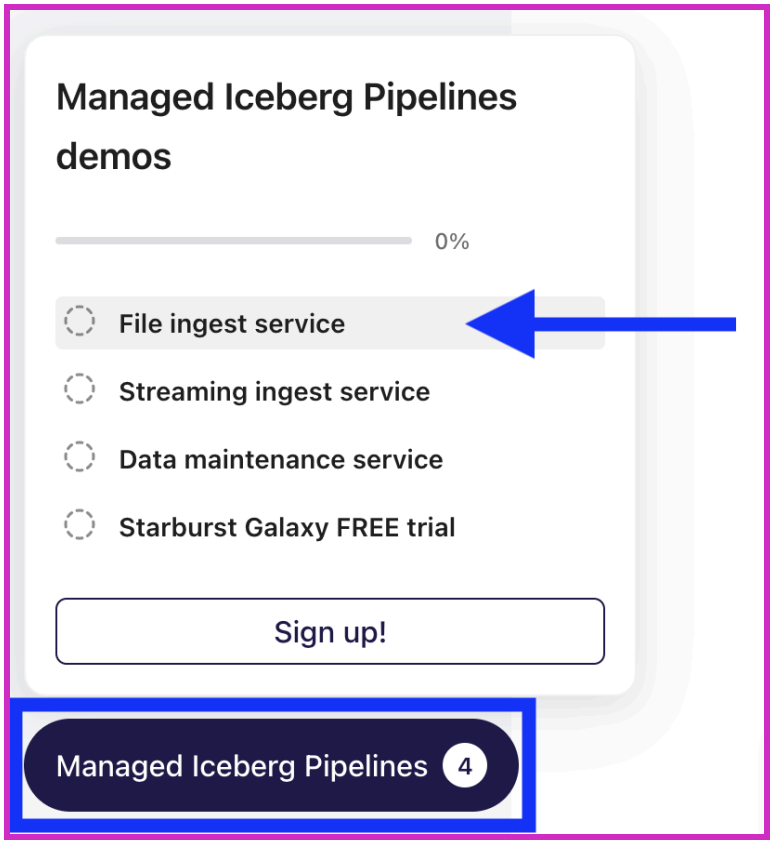
- Sign in and verify the `students` role is selected in the upper-right corner.
- Ensure the `aws-us-east-1-small` **Cluster** is reporting a **Status** of `Running`.
- In the **Query editor**, select `aws-us-east-1-small` in the cluster pulldown.

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.**

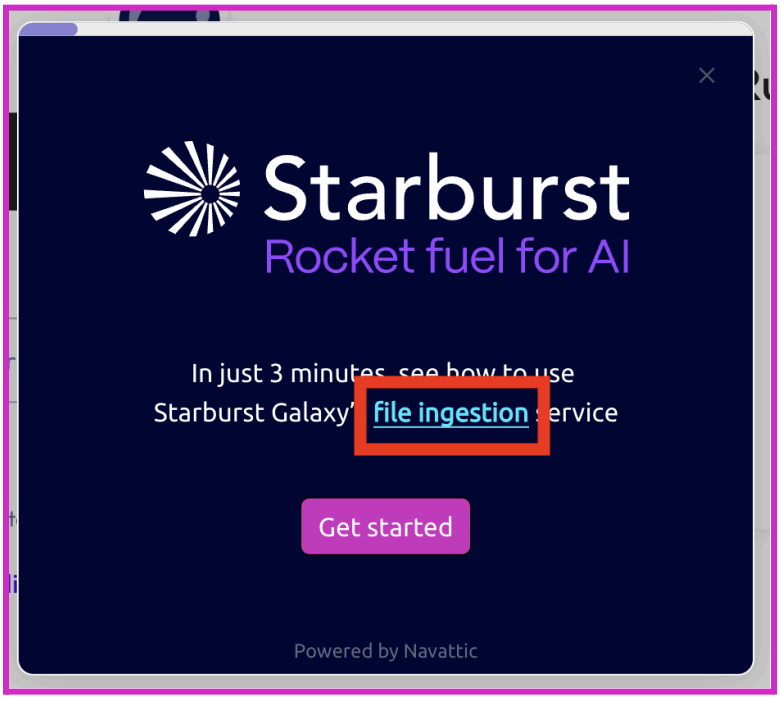
```
CREATE SCHEMA students.yourname;
```

Step 2 - Locate the interactive demo instructions

Open your browser to https://starburst.navattic.com/kfa079f, click on the **Managed Iceberg Pipelines** button in the lower-left corner, and click on **File ingest service**.



Verify a modal box is present in the center of your screen displaying it is for the **file ingestion** demonstration.



Step 3 - Perform demo steps

Press the **Get started** button and use the interactive instructions to guide you through setting up and utilizing this service. Use the following specific values when prompted.

Demo step 5 of 21

Set **Source name** to `yourname_file_ingest_source`. Make sure that you replace `yourname` with the name you used for your schema.

Demo step 6 of 21

Set **S3 bucket** to `starburst101-handsonlab-nyc-uber-rides` and continue to use `file_ingest` for the **S3 file prefix** value.

Demo step 7 of 21

For **Authenticate with**, choose the **Cross account IAM role** radio button then select **edu-train-role** from the **Cross account IAM role** pulldown.

Demo step 11 of 21

Make the following selection in the Raw table target section.

- Choose **students** from **Catalog** pulldown
- Choose ***YOUR schema*** from the **Schema** pulldown
- Enter `raw_spawn_file` for **Table name**

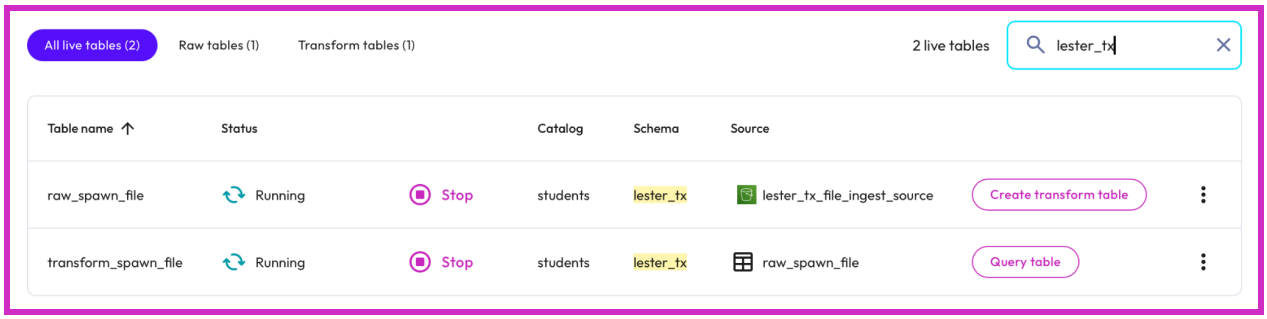
Demo step 15 of 21

Enter `transform_spawn_file` for **Table name**.

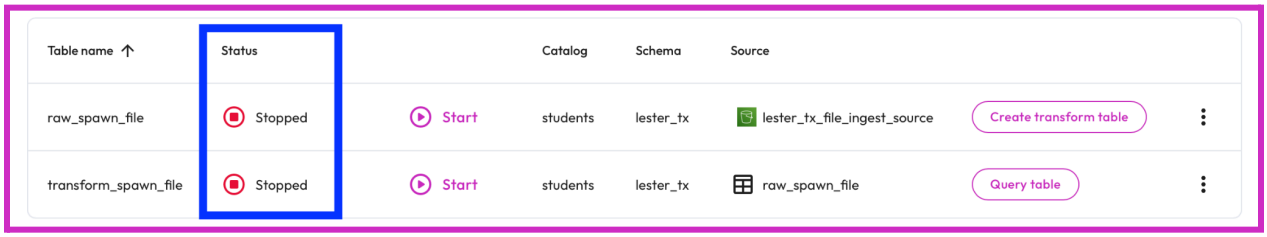
Step 4 - Stop live tables

Return to **Data** > **Data ingest** and scroll down to the **Live tables** section. Enter ***YOUR schema*** in the search box whose value defaults to **Search live tables**.

Only your live tables will be present now.



For each of YOUR tables with a **Status** of **Running**, click on the **Stop** link. Verify all get their **Status** updated to **Stopped**.



END OF LAB EXERCISE

Lab 2: Explore the streaming ingestion service (10 mins)

Learning objectives

- In this lab, you will configure and execute the streaming ingestion service.

Prerequisites

- Getting started - Lab 1: Create student account

Activities

1. Prepare for the exercise
2. Locate the interactive demo instructions
3. Perform demo steps
4. Stop live tables

Step 1 - Prepare for the exercise

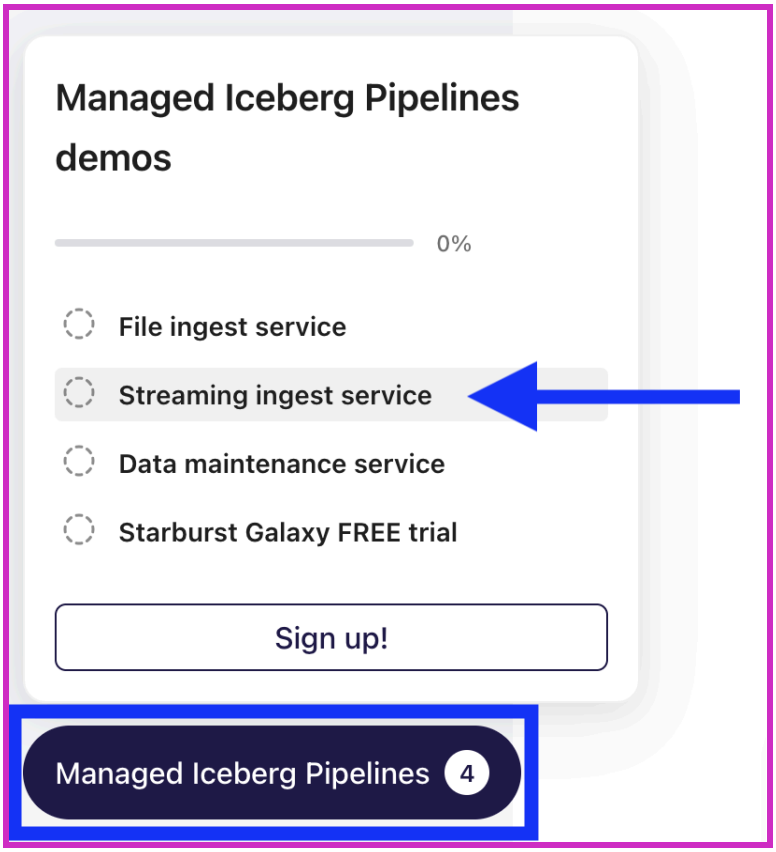
- Sign in and verify the `students` role is selected in the upper-right corner.
- Ensure the `aws-us-east-1-small` **Cluster** is reporting a **Status** of `Running`.
- In the **Query editor**, select `aws-us-east-1-small` in the cluster pulldown.

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.**

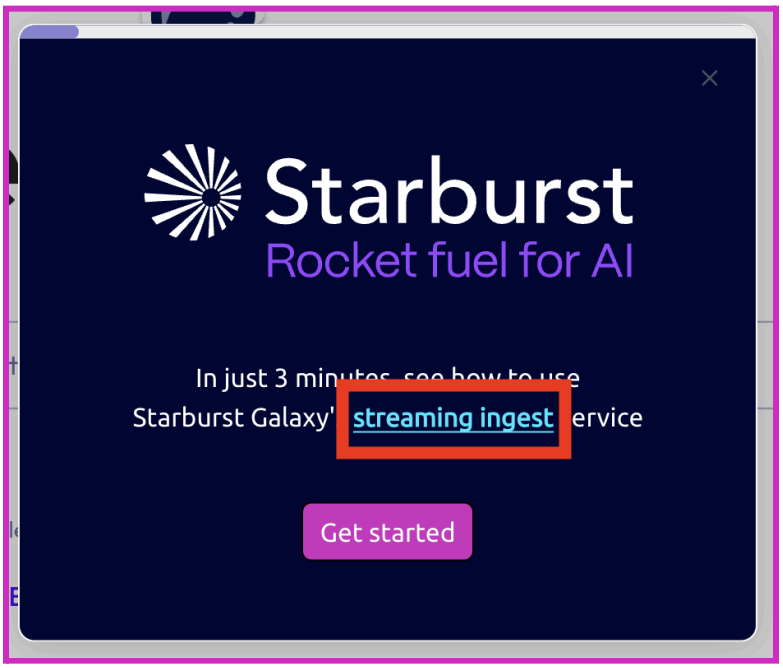
```
CREATE SCHEMA students.yourname;
```

Step 2 - Locate the interactive demo instructions

Open your browser to https://starburst.navattic.com/kfa079f, click on the **Managed Iceberg Pipelines** button in the lower-left corner, and click on **Streaming ingest service**.



Verify a modal box is present in the center of your screen displaying it is for the **streaming ingestion** demonstration.



Step 3 - Perform demo steps

Press the **Get started** button and use the interactive instructions to guide you through setting up and utilizing this service. Use the following specific values when prompted.

Demo step 5 of 21

Set **Source name** to `yourname_streaming_ingest_source`. Make sure that you replace `yourname` with the name you used for your schema.

Demo step 6 of 21

Set **Kafka brokers** to `pkc-p11xm.us-east-1.aws.confluent.cloud:9092`.

Demo step 7 of 21

For **Authenticate with**, choose the **API Key / API Secret** radio button then select **SASL/PLAIN** from the **Auth mechanism** pulldown.

Enter `7OZFGK6TBTZEGEBN` for **API key**.

Enter `cFU4421Q6l7vdLoBqdvBSM5wvDL9I6LbuTabE/HUOM0RwbH8+CQriifdxL4hieNN` for **API secret**.

Demo step 11 of 21

Make the following selection in the Raw table target section.

- Choose **students** from **Catalog** pulldown
- Choose ***YOUR schema*** from the **Schema** pulldown
- Enter `raw_spawn_stream` for **Table name**

Choose **Start from earliest messages** from the **Streaming ingest start point** pulldown.

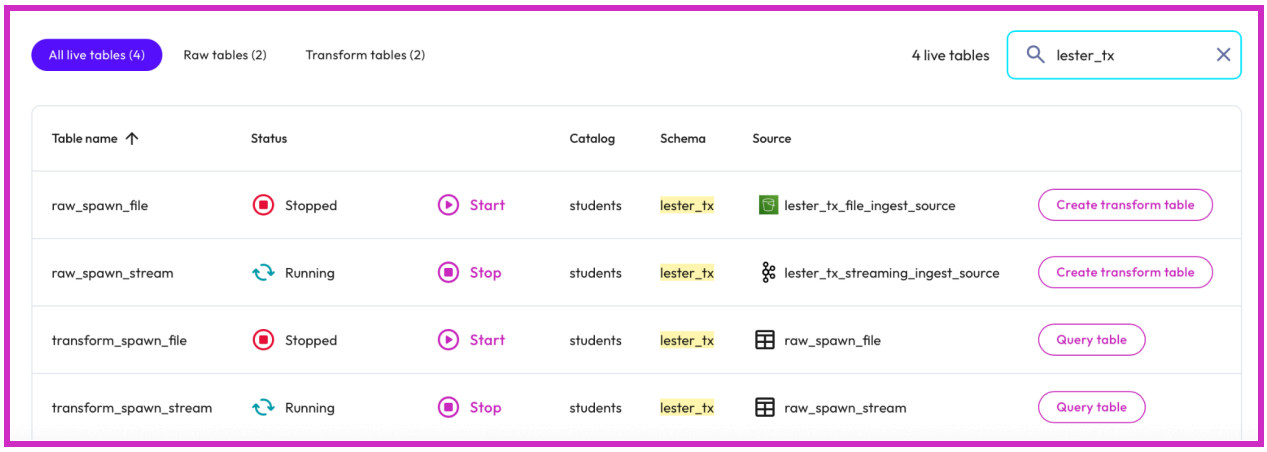
Demo step 15 of 21

Enter `transform_spawn_stream` for **Table name**.

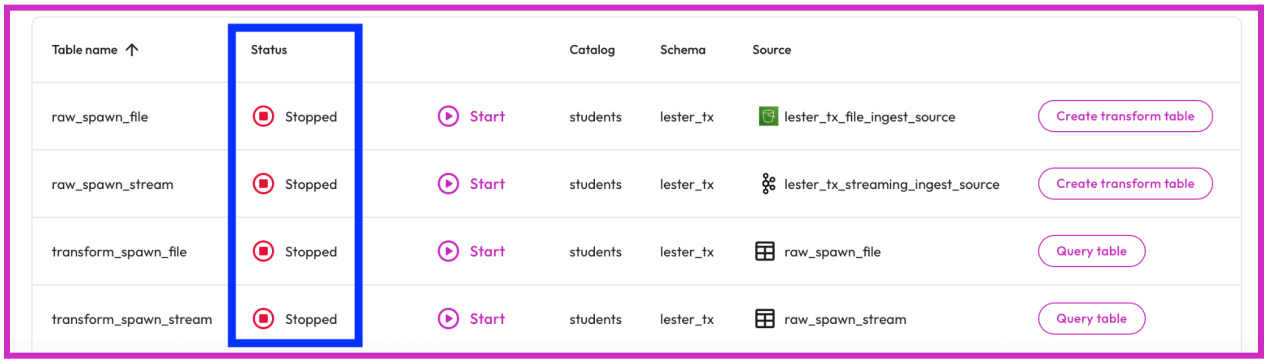
Step 4 - Stop live tables

Return to **Data** > **Data ingest** and scroll down to the **Live tables** section. Enter ***YOUR schema*** in the search box whose value defaults to **Search live tables**.

Only your live tables will be present now.



For each of YOUR tables with a **Status** of **Running**, click on the **Stop** link. Verify all get their **Status** updated to **Stopped**.



END OF LAB EXERCISE