Trino and Starburst Training Series: Creating & querying data lake tables

v1.0.0

Session 1 of 5: Full Series Information

This workshop is focused on creating & querying data lake tables. These are the goals for this session.

- Leverage Starburst Galaxy to create a catalog and schema aligned to an AWS S3 object store.
- Construct external tables to existing datasets.
- Utilize optimized columnar file formats to improve performance.
- Design partitioned tables to improve performance.
- Federate data lake queries to join with traditional data sources.

Table of Contents

Lab 1: Create Starburst Galaxy account and a data lake catalog (15 mins)

Learning objectives

- This lab will walk you through the process of creating a new account within Starburst Galaxy. You will create a domain name and password and set up your account to begin using sample data. A data lake catalog, and associated privileges, will be created. Finally, you will connect this new catalog to an existing cluster.

Prerequisites

- None.

Activities

1. Sign up for Starburst Galaxy (one-time event)
2. Configure a data lake catalog
3. Set permissions
4. Add to cluster
5. Grant location-based access control
6. Start the cluster

Step 1 - Sign up for Starburst Galaxy (one-time event)

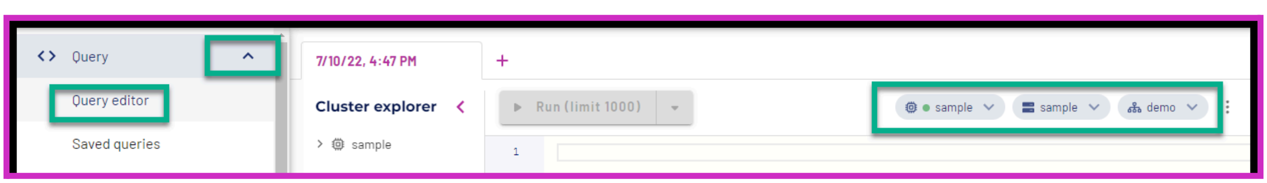
For each webinar in the 5-part series, you will be using your own Starburst Galaxy environment.
If you have already registered for Starburst Galaxy, you may skip this step.

To sign up for Starburst Galaxy, follow the instructions on the free registration page at https://www.starburst.io/platform/starburst-galaxy/start/.

Note: When prompted, choose to **not** connect your data sources, but choose to use the sample data.

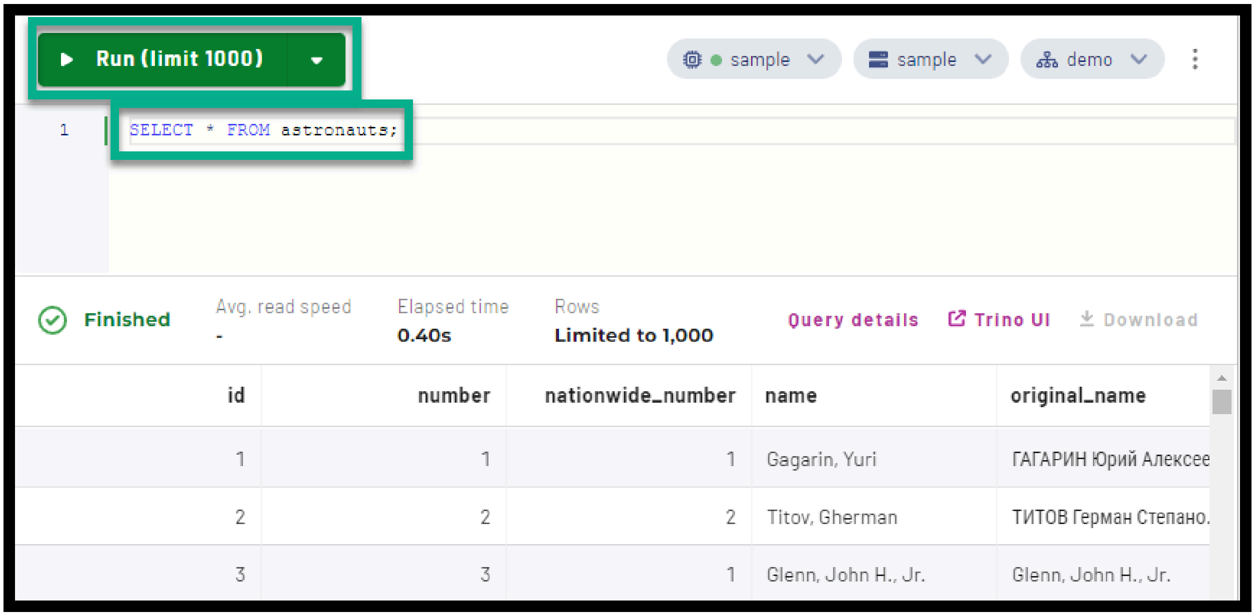
Follow these steps to open the **Query editor**.

- Expand **Query** from the left menu
- Click **Query editor**
- In the **Select cluster** drop down, select **free-cluster**
- In the **Select catalog** drop down, select **sample**
- In the **Select schema** drop down, select **demo**



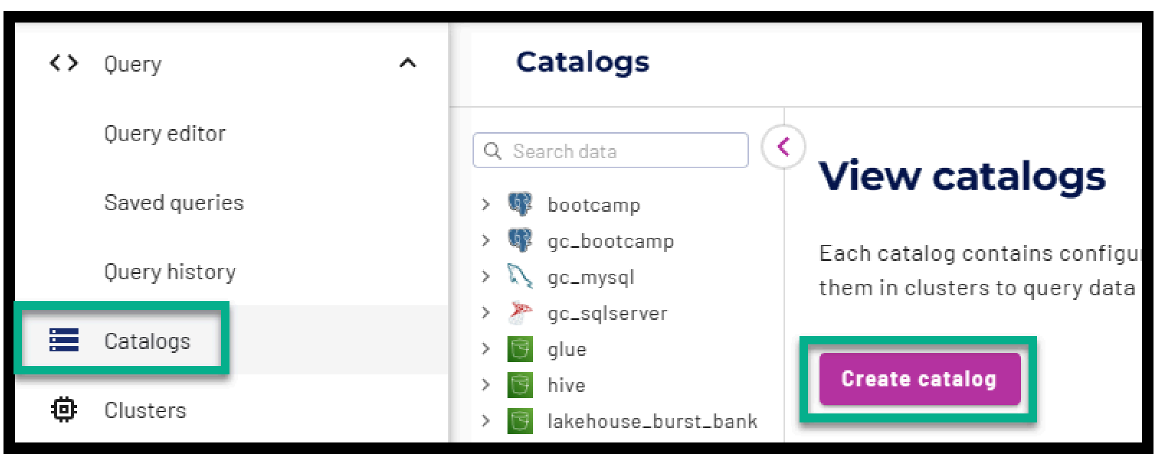
Paste the following SQL into the editor and then click **Run (limit 1000)**.

```
SELECT * FROM astronauts;
```



Step 2 - Configure a data lake catalog

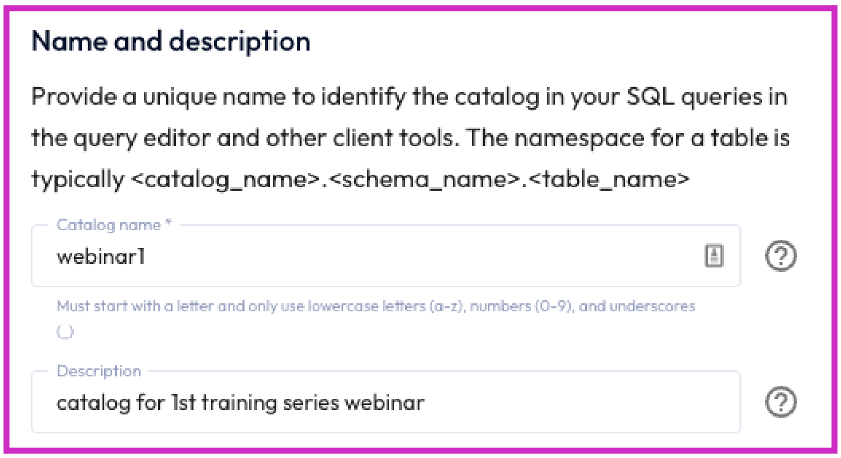
Click **Catalogs** in the menu on the left and then click the **Create catalog** button.



Click the **Amazon S3** tile.

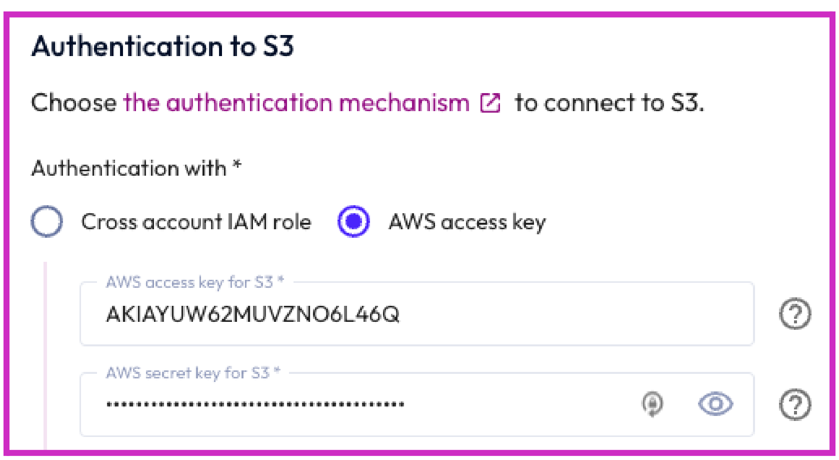
Use the information below to configure your catalog.

Catalog name: `webinar1`
Description: `catalog for 1st training series webinar`

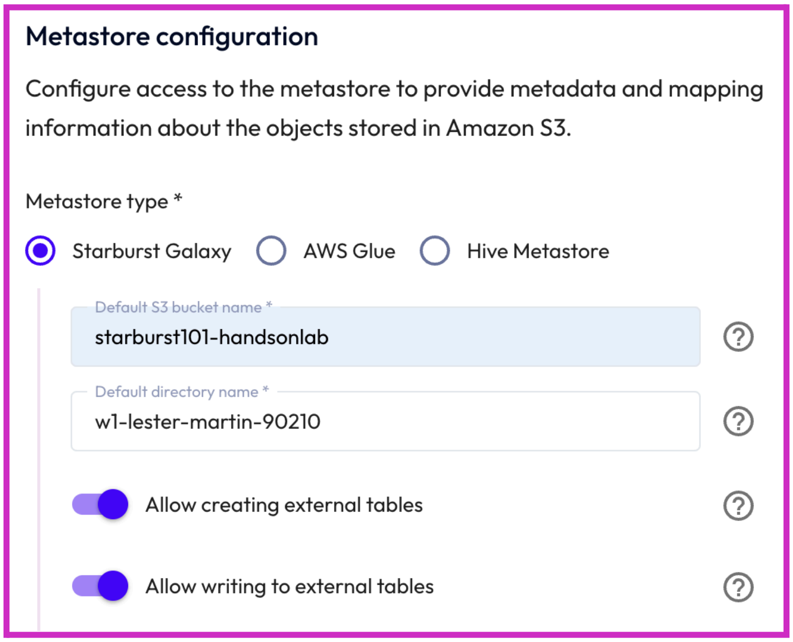


Authentication with: select the radio button **AWS access key**
AWS access key for S3: `AKIAYUW62MUVSUZ6LKRG`
AWS secret key for S3: `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`

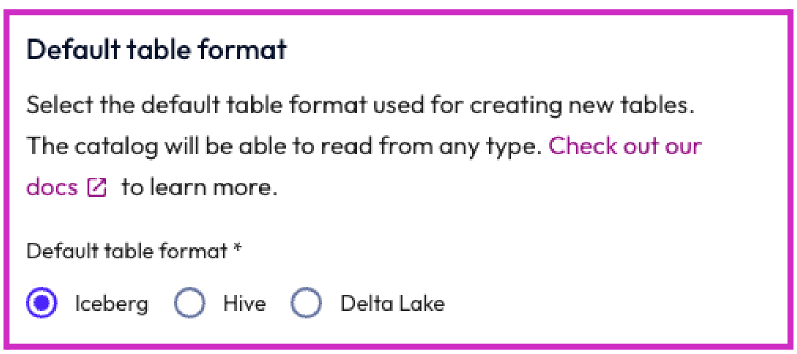
Note: These AWS credentials will only be operational through the weekend following the webinar. This means you will not be able to utilize this catalog beyond that point.



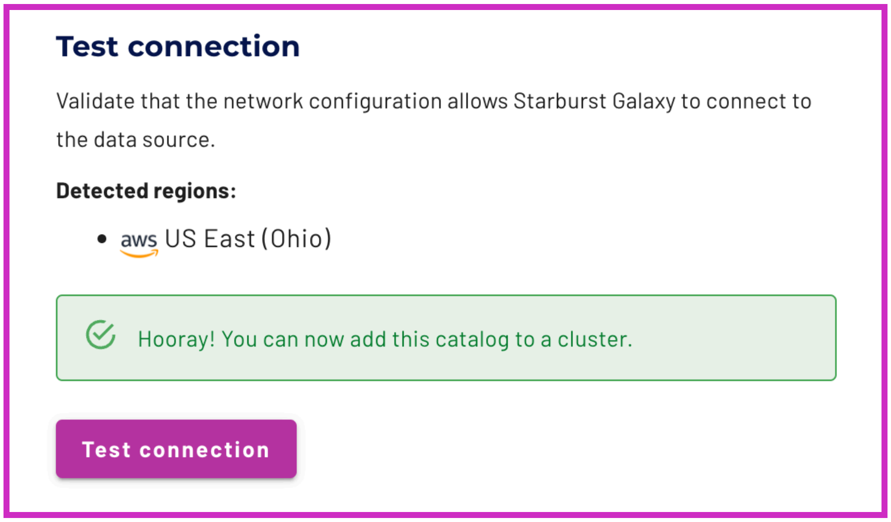
Metastore type: select the radio button **Starburst Galaxy**
Default S3 bucket name: `starburst101-handsonlab`
Default directory name: w1-fname-lname-postalcode (ex: `w1-lester-martin-90210`)
    Allow creating external tables: enable the slider
    Allow writing to external tables: enable the slider



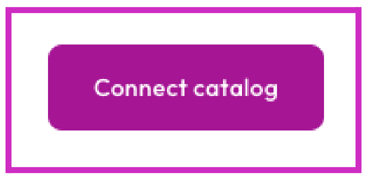
Default table format: ensure the radio button is selected to **Iceberg**



Validate the connection by hitting **Test connection.** Your catalog should return the same message indicating that you can now add the catalog. Confirm you see the **Hooray! You can now add this catalog to a cluster** message.

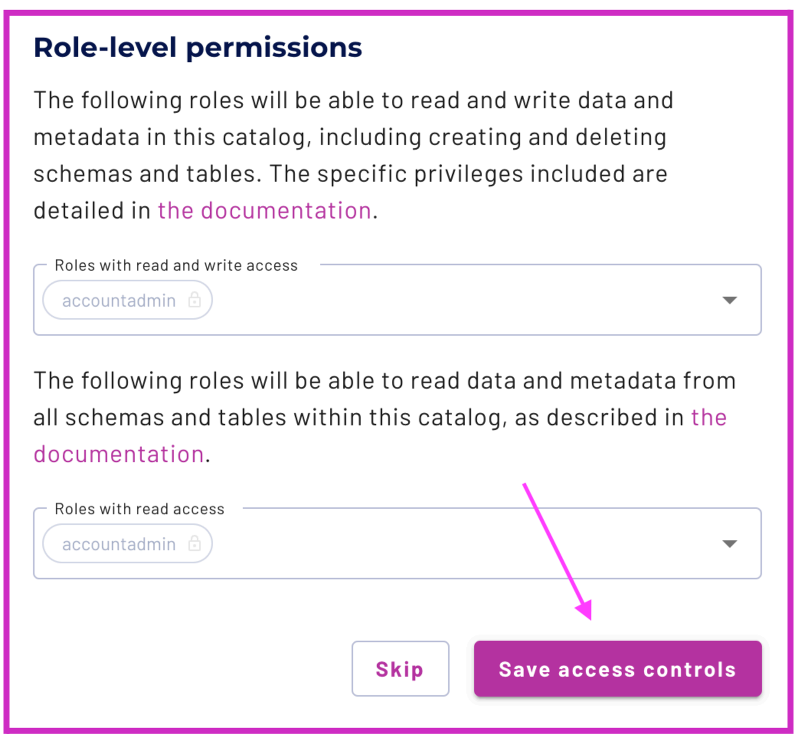


Select **Connect catalog**. This will save the credentials for your Amazon S3 catalog.



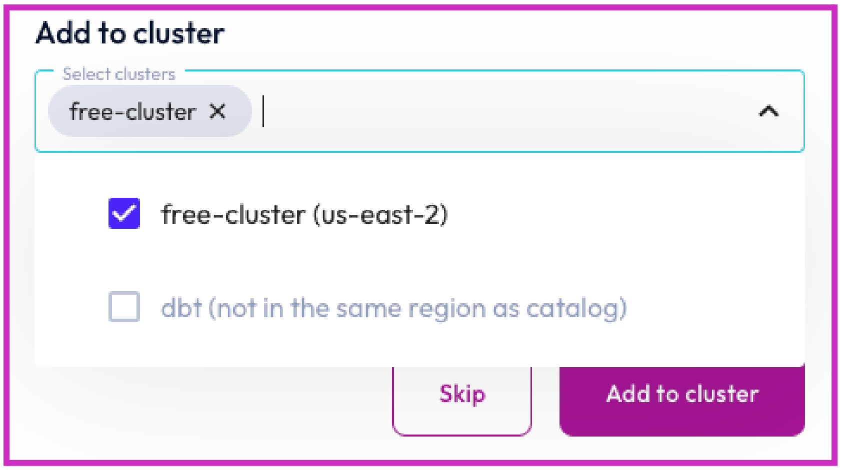
Step 3 - Set permissions

Next, accept the default permissions for your catalog by selecting the button **Save access controls**.

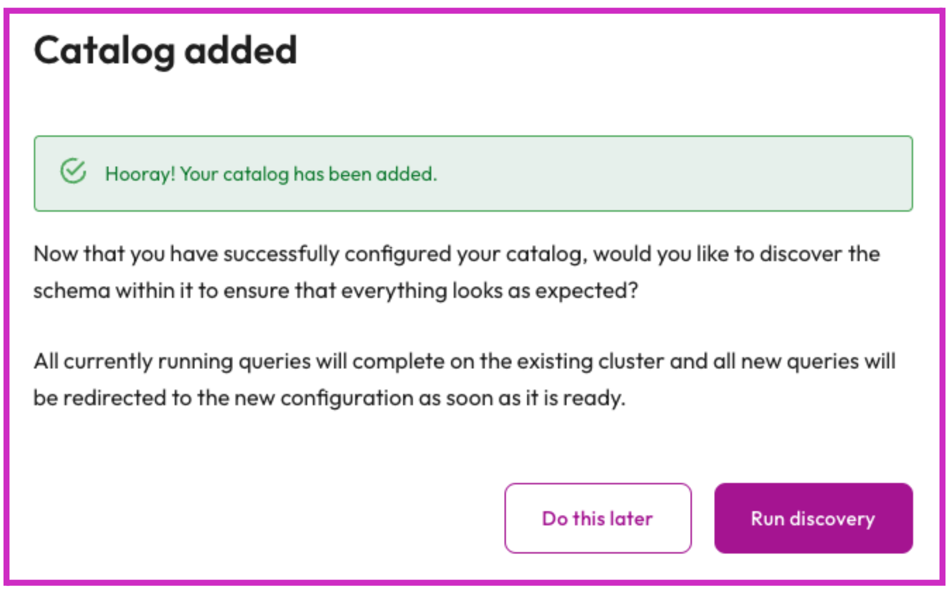


Step 4 - Add to cluster

Select `free-cluster` in the **Select clusters** pulldown and then click on **Add to cluster**.

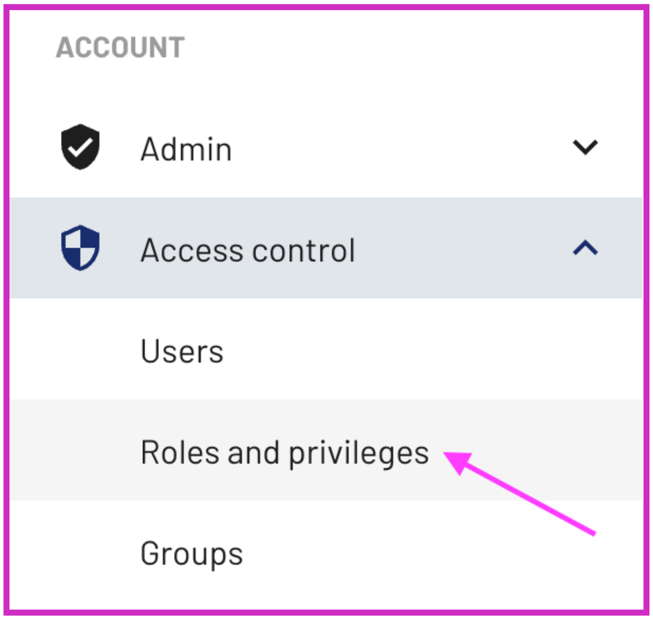


Click **Do this later** in the **Catalog added** pop-up.



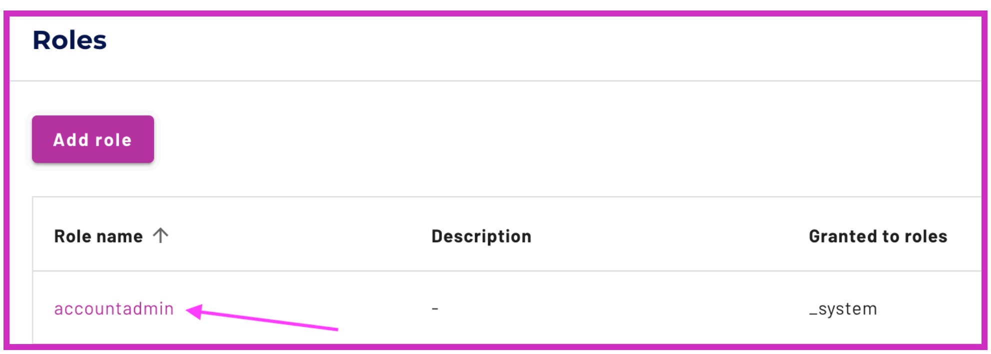
Step 5 - Grant location-based access control

Navigate to the **Roles and privileges** section under **Access control** on the left-hand side of Starburst Galaxy.

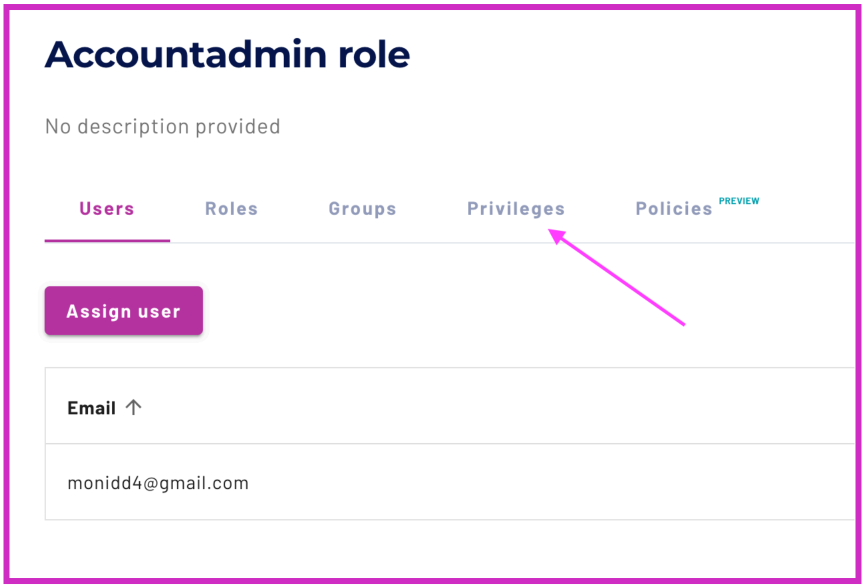


Starburst Galaxy's built-in access control allows you to define multiple users, roles, groups, privileges, and policies. These access options encourage businesses to create security policies that make sense for their organization.

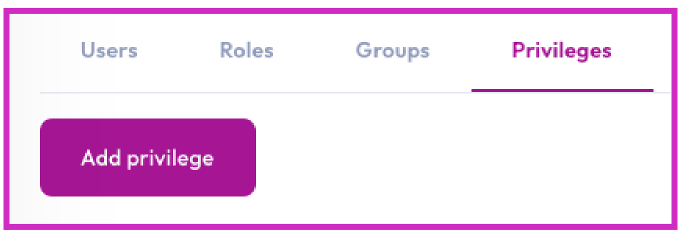
Click the **accountadmin** role link to add additional privileges.



Navigate to the **Privileges** tab to add location-based access.

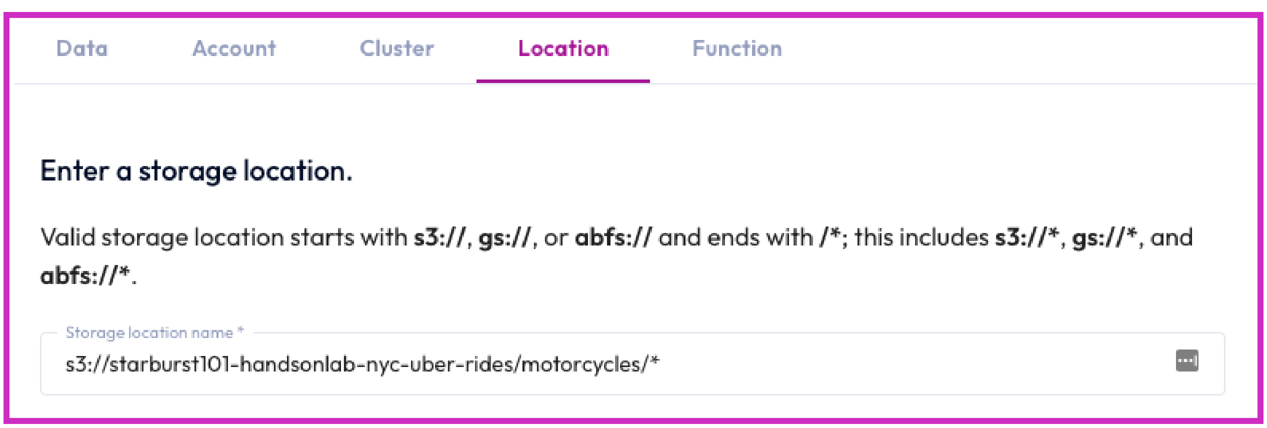


Select the **Add privilege** button.



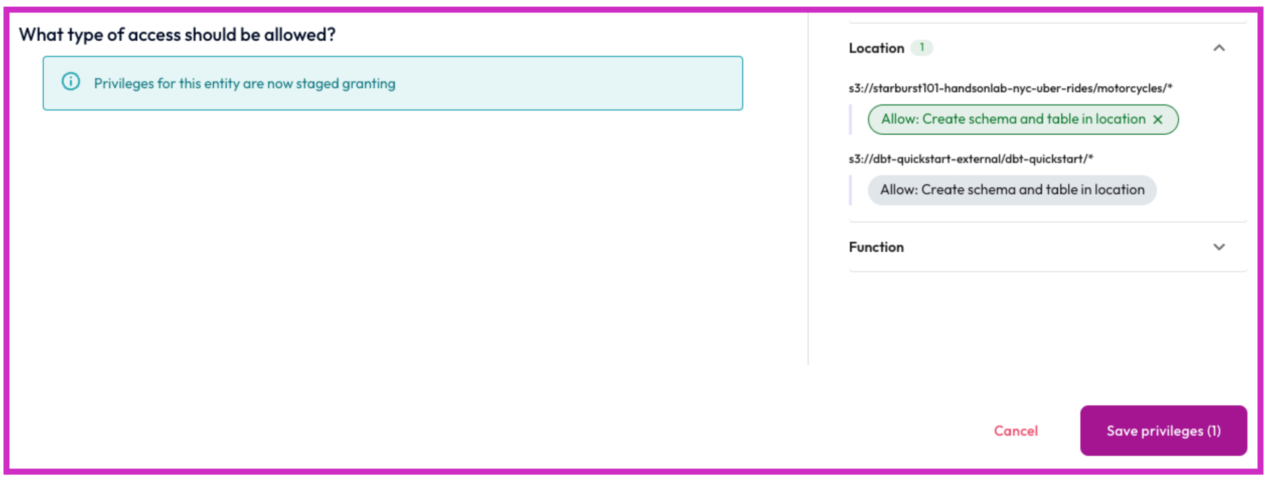
Select the **Location** tab and add the **Storage location name** value below. This provides access to a location in S3 and allows you to create an external table from this location.

```
s3://starburst101-handsonlab-nyc-uber-rides/motorcycles/*
```



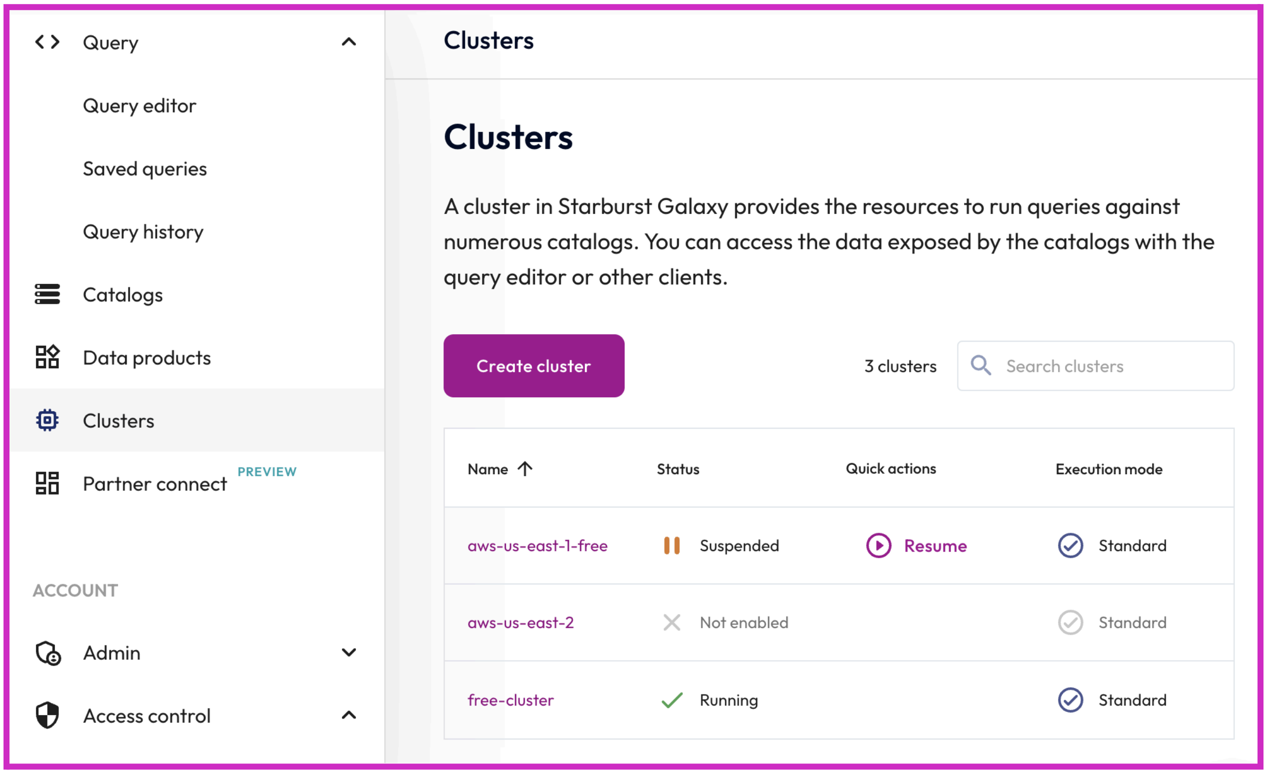
Check the **Create schema and table in location** checkbox that surfaces.

After the checkbox is selected, a **Privileges for this entity are now staged granting** message replaces it and is highlighted in blue. Click the **Save privileges** button in the lower right.



Step 6 - Start the cluster

Click on **Clusters** on the left menu to see a list of the configured clusters in your account. Likely, you will only have one named `free-cluster`. The screenshot below shows multiple clusters.



If the **Status** for `free-cluster` is **Running**, then you are done with this step. If the **Status** is **Suspended**, click on **Resume** under **Quick actions** and wait for it to report as **Running**.

END OF LAB EXERCISE

Lab 2: Construct an external table leveraging existing datasets (10 mins)

Learning objectives

- In this lab you will be presented with definitions for the data lake itself and the data lake tables that leverage it. You will review a few data files and then construct a table pointing to them. You will query the table with standard SQL. Lastly, you will learn what an external table is.

Prerequisites

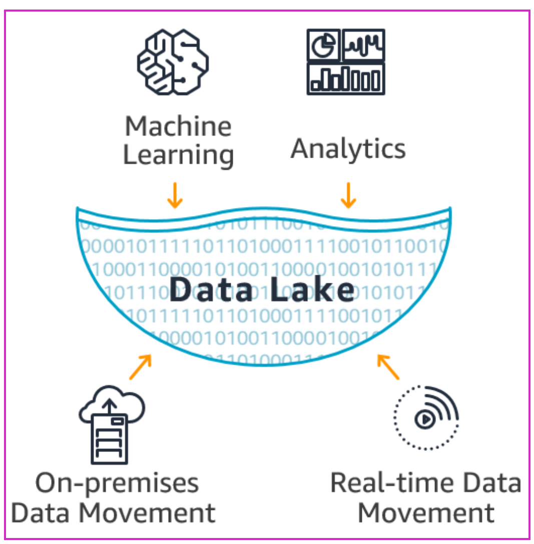
- [Lab 1 - Create Starburst Galaxy account and a data catalog]

Activities

1. Define the data lake
2. Explore the datasets
3. Understand data lake tables
4. Construct and query a table
5. Explain external tables

Step 1 - Define the data lake

A "data lake" is a repository of data that can store files in any type of format; including in its natural/raw format. It allows for economical decoupled storage that can be reused for multiple purposes by multiple engines & frameworks.



For this workshop, the underlying technology used for our data lake is Amazon S3.

Step 2 - Explore the datasets

There is a folder on Amazon S3 housing a few files that contain motorcycle information.

Amazon S3 > Buckets > starburst101-handsonlab-nyc-uber-rides > motorcycles/ > csv_files/

NOTE: You do not have access to view this S3 "bucket" directly, but here are the files that are in that directory.

Name	Type	Last modified	Size
even_more.silly.ext	ext	January 24, 2024, 19:27:04 (UTC-05:00)	81.0 B
more_motorcycles	-	January 24, 2024, 19:27:04 (UTC-05:00)	81.0 B
motorcycles.csv	csv	January 24, 2024, 19:27:04 (UTC-05:00)	175.0 B

Here is the contents of the file named `motorcycles.csv`.

```
BMW,R1150RS,2004,14274
Kawasaki,GPz1100,1996,60234
Ducati,ST2,1997,24000
Moto Guzzi,LeMans,2001,12393
BMW,R1150R,2002,17439
Ducati,Monster,2000,15682
Aprilia,Futura,2001,17320
```

This CSV file does not contain a header row, but upon reviewing it you might notice it appears to be a list of used motorcycles, likely for sale, that has these fields for each record.

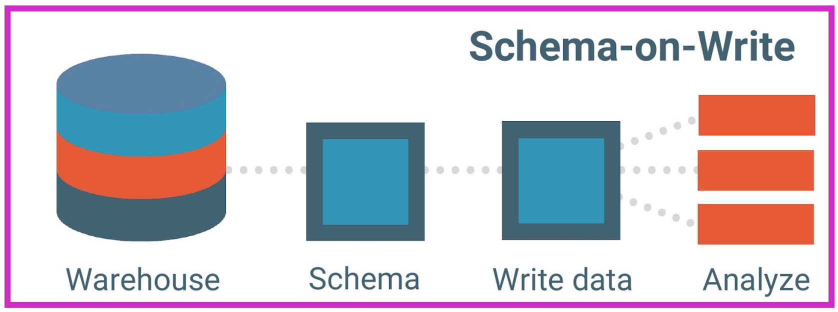
- Make
- Model
- Year
- Number of miles (or kilometers)

While this file has an extension of `.csv`, this is not required for our eventual querying of this information. Note the file names of the other two files, `more_motorcycles` (no extension) and `even_more.silly.ext` (a purposely misleading file extension), which have these contents.

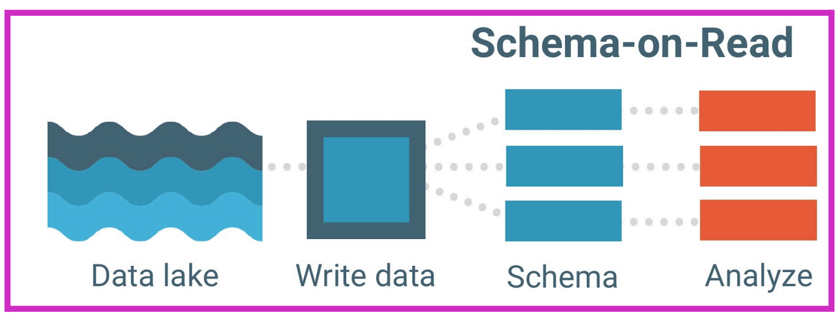
more_motorcycles	even_more.silly.ext
Suzuki,GS500,2002,82918 Honda,CBR1000RR ABS,2022,6628 KTM,Freeride E-XC,2022,1205	Honda,Gold Wing Tour 1800,2018,17453 Harley-Davidson,Sportster Iron 883,2017,4838

Step 3 - Understand data lake tables

Traditional data warehouses require a pre-defined schema before data can be loaded. This strategy is called *Schema on Write* as data must be fully transformed to match the data model before it can be loaded.



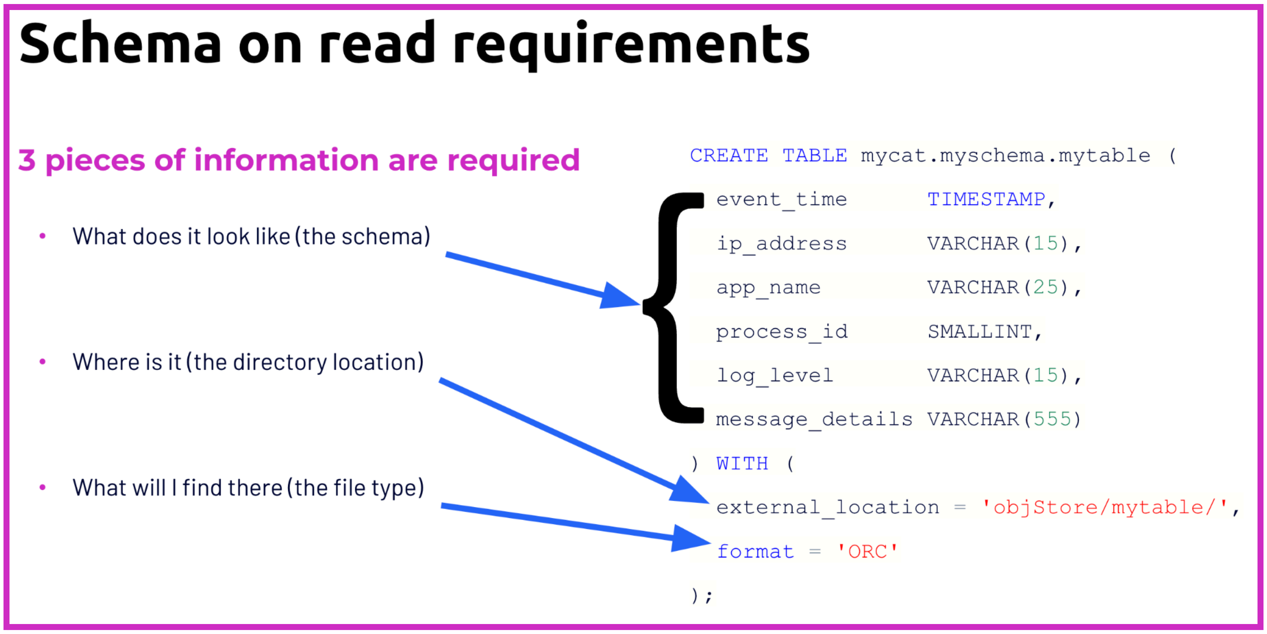
Conversely, data lake tables utilize a model called *Schema on Read*. Data is loaded in its raw form. The table definition needs only to be created before querying the data. At that time, the data on the lake is aligned with the schema and results are presented.



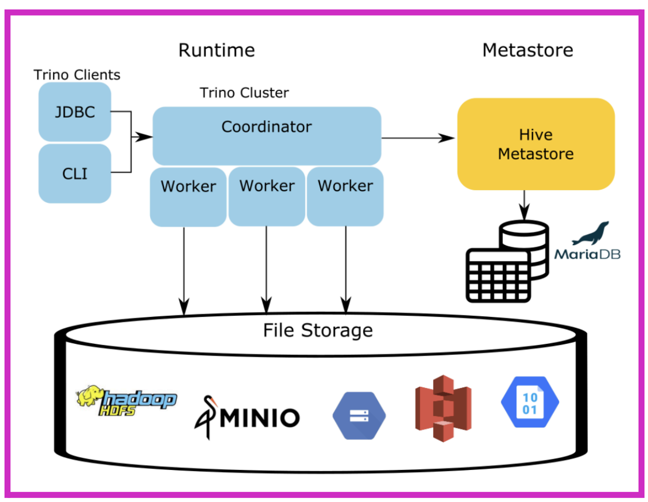
We have already loaded the raw motorcycles files into the data lake. Now we need to create a table that we can run queries against. The metadata that is stored for a data lake table contains 3 important elements.

1. What does it look like (the schema itself)
2. Where is the data (the directory location of the files)
3. What file format is being utilized (such as CSV, Avro, or Parquet)

Here is an an example of how to declare these 3 important elements.

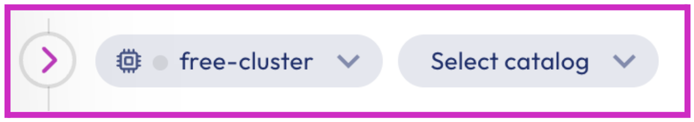


This table-defining information is stored in a metastore service. The following diagram shows that queries come into the Trino/Starburst cluster which in turn interrogates the metastore for the table configuration, and finally reads the data from the lake to complete the query.



Step 4 - Construct and query a table

Select **Query** and **Query editor** from the left menu and then make sure `free-cluster` is the selected cluster.



Paste and **Run** the following SQL into a new editor tab.

```
CREATE SCHEMA webinar1.myschema;
USE webinar1.myschema;
```

This created a schema that you can create tables within. It also updates the pulldowns to the right of the cluster name to your newly created schema.

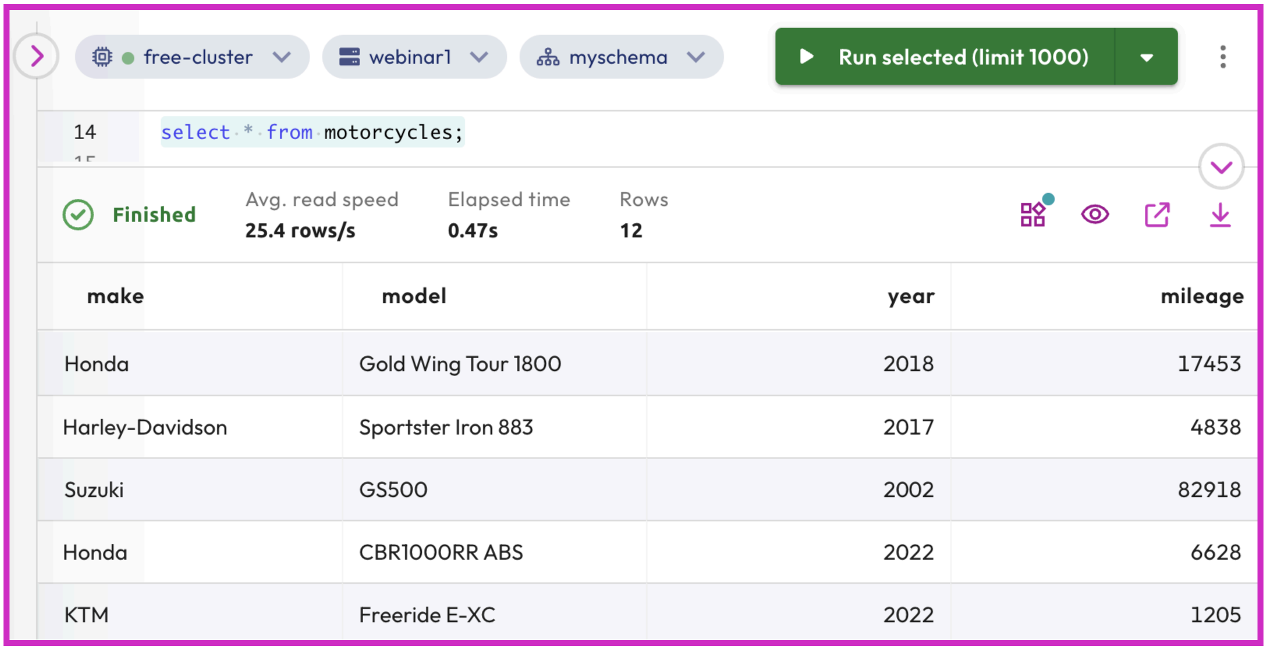
Execute the following SQL to create a table aligned with the CSV files we reviewed earlier.

```
CREATE TABLE motorcycles (
  make varchar,
  model varchar,
  year int,
  mileage int
) WITH (
  external_location =
    's3://starburst101-handsonlab-nyc-uber-rides/motorcycles/csv_files/',
  type = 'hive', format = 'TEXTFILE', textfile_field_separator = ','
);
```

All 3 elements of a data lake table are present in the above Data Definition Language (DDL) SQL statement..

1. The schema
2. The S3 directory
3. CSV file format

Run a `SELECT *` query on the new table.



Notice that 12 rows were returned which are the number of rows from the 3 underlying files on S3 reviewed earlier.

Run the following query to verify that standard SQL queries work on data lake tables.

```
SELECT year, format('%,.0f', avg(mileage)) AS avg_mileage
  FROM motorcycles
 WHERE year > 2009
 GROUP BY year
 ORDER BY year;
```


Step 5 - Explain external tables

In this approach where the data files are already present prior to the table being created, we can load more and more data by simply placing additional files in the data lake directory identified in the DDL.

This approach allows other applications & frameworks to create data that can then be queried by Starburst Galaxy, as well as other SQL engines. This is referred to as an “external table” which can be identified by an `external_location` property in the `WITH` clause of the `CREATE TABLE` statement.

The alternative to an external table is just called a “table” – or just as often, a “managed table”. You will create these types in subsequent labs and some additional information will be provided about how they work.

For now, the most important thing to know is that if you `DROP` an external table, you only delete the metadata. The underlying data on the data lake remains. With the managed table approach, the `DROP` deletes the metadata as well as the underlying data.

END OF LAB EXERCISE

Lab 3: Improve performance with columnar file formats (20 mins)

Learning objectives

- After learning the differences between row-oriented and columnar file formats, you will explore how a managed table works. You will create ORC and JSON tables with the same logical characteristics and load them with the same actual data. You will compare the underlying file sizes of ORC and JSON tables. Lastly, you will run multiple performance comparisons between the two to verify the columnar file formats are far superior to the simple row-oriented file formats.

Prerequisites

- Lab 1 - Create Starburst Galaxy account and a data catalog

Activities

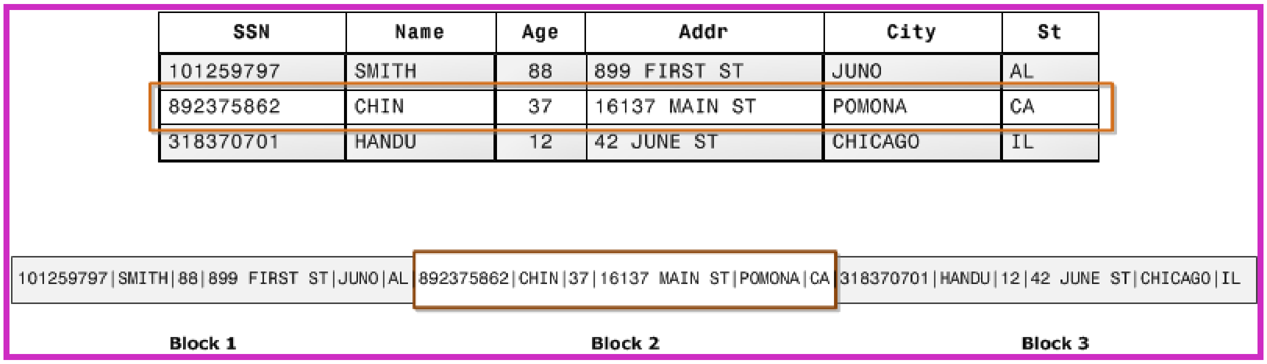
1. Understand the file format options
2. Limiting the size of data read with columnar formats
3. Create & populate an ORC-based managed table
4. Construct tables for performance tests
5. Compare performance for table row count
6. Compare performance with projection & filtering query
7. Compare performance while calculating aggregation values

Step 1 - Understand the file format options

In this workshop we are leveraging what are called Hive tables. This table format is the original data lake approach and it has the most flexibility in the type of files it can support. Visit the full list of supported file formats to learn more.

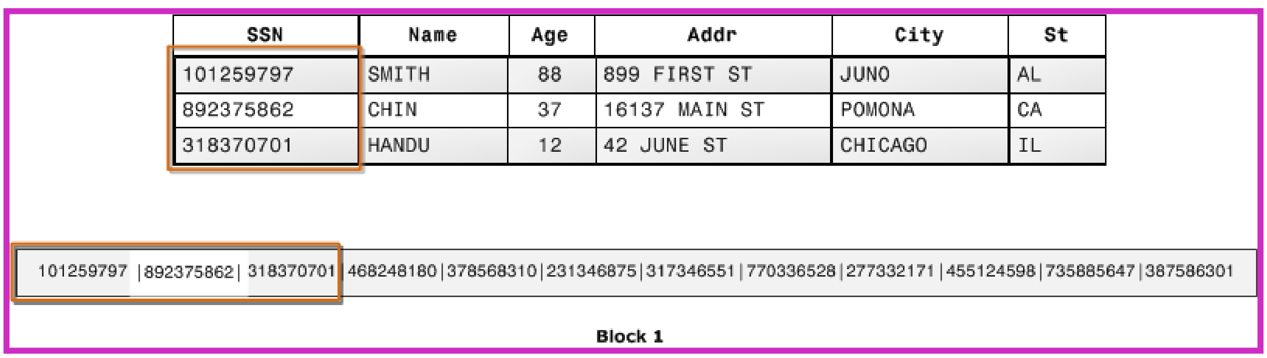
There are several different file formats available, but they generally break down into two types of files; row-oriented formats & columnar file formats.

The easiest to understand is the row-oriented format. All data for a record is kept together and written in the order of the fields. Subsequent records are appended to the end of the previous record. This is visualized below.



Popular row-oriented file formats include CSV, JSON, and Apache Avro.

The more optimized columnar file formats are a bit harder to conceptualize. Data for a specific column (across all records) is stored independently from other columns. Low cardinality fields may employ a dictionary of unique values thereby shrinking the physical size of the data. A visualization of this concept (only focusing on the first column) follows.

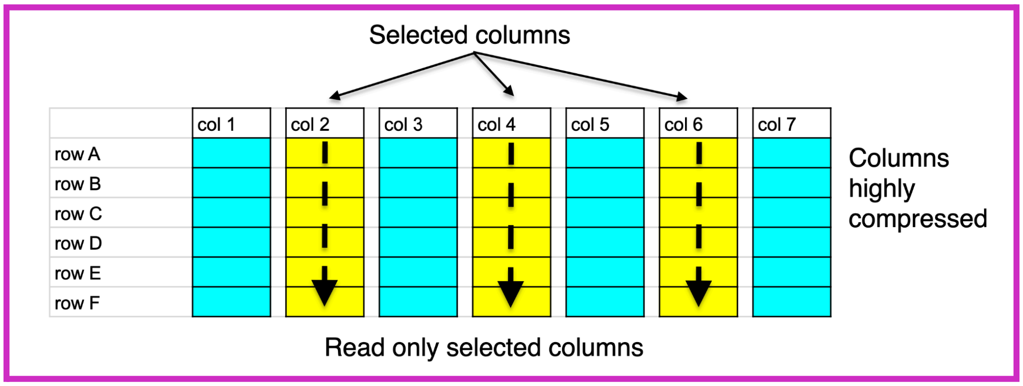


The writing of these files is more complex than row-oriented since the framework creating these has to wait until a large number of logical rows are known before it can then physically write the separate columns that include all the row's data for each column. The column file formats take the opportunity to include in the actual files lots of column-level statistics that will be leveraged in analytical queries such as `GROUP BY` and `WINDOW` functions.

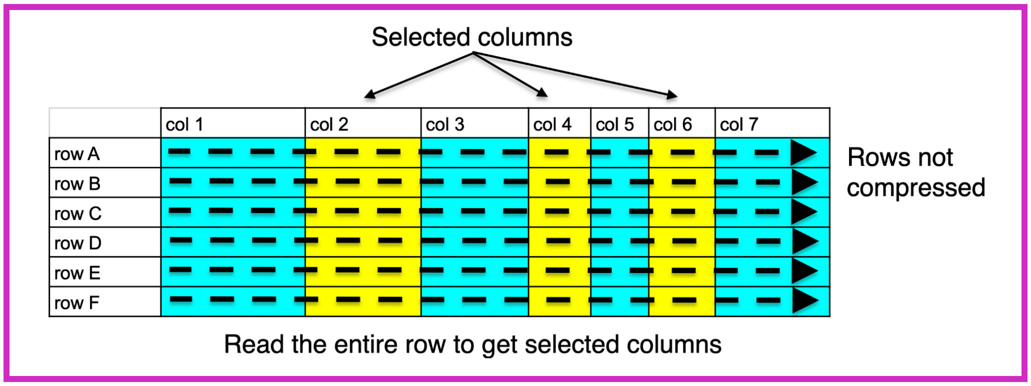
Apache Parquet and Apache ORC are the widely accepted implementations of columnar file format for data lake analytics.

Step 2 - Limiting the size of data read with columnar formats

One of the easiest benefits of these file formats to visualize is when not all columns are requested in a query. Columnar file formats allow data to be pulled in from only the needed columns.



Row-oriented file formats have to read all data for each row from the data lake and then only once being processed in memory can these unwanted columns be ignored.



In data lake analytics, data movement is one of the highest, if not the most, expensive operations in an overall query execution. Reading 3 columns out of 7 present suggests approximately 50% less data was read, but imagine if you only need 3 columns of of 77 (or even 777). You will see incredible improvements in these scenarios.

Step 3 - Create & populate an ORC-based managed table

Run the following DDL to create a managed table utilizing the ORC file format.

```
CREATE SCHEMA IF NOT EXISTS webinar1.myschema;
USE webinar1.myschema;

CREATE TABLE dune_characters (
   id integer,
   name varchar(55),
   affiliation varchar(55),
   spacing_guild_ktn varchar(55), -- Known Traveler Number
   notes varchar(255)
) WITH ( type = 'hive', format = 'orc');
```

You might have noticed that there was no `external_location` property in the `WITH` clause which by definition means it is not an external table, but a managed one. In this case, the location for the table's datasets becomes a folder on the data lake underneath the schema's folder it belongs to.

Run the following SQL to load some data and verify it was inserted into the table.

```
INSERT INTO dune_characters
(id, name, affiliation, spacing_guild_ktn, notes)
VALUES
(101, 'Leto Atreides', 'House Atreides', '15KW1353W', 'Duke'),
(102, 'Lady Jessica', 'House Atreides', '15KW1353X', 'Concubine of the Duke'),
(103, 'Paul Atreides', 'House Atreides', 'TT782N33U', 'Son of Leto'),
(104, 'Thufir Hawat', 'House Atreides', '10397236Y', 'Mentat'),
(105, 'Stillgar', 'Fremen', '15KW1353W', 'Naib of Stietch Tabr'),
(106, 'Shadout Mapes', 'Fremen', 'TTKW898JD', 'Head housekeeper'),
(107, 'Gaius Helen Mohiam', 'Bene Gesserit', '5078872FY', 'Reverend Mother');

SELECT * FROM dune_characters;
```

Finished | Avg. read speed 7.9 rows/s | Elapsed time 0.89s | Rows 7

id	name	affiliation	spacing_guild_ktn	notes
101	Leto Atreides	House Atreides	15KW1353W	Duke
102	Lady Jessica	House Atreides	15KW1353X	Concubine of the Duke
103	Paul Atreides	House Atreides	TT782N33U	Son of Leto
104	Thufir Hawat	House Atreides	10397236Y	Mentat
105	Stillgar	Fremen	15KW1353W	Naib of Stietch Tabr
106	Shadout Mapes	Fremen	TTKW898JD	Head housekeeper
107	Gaius Helen Mohiam	Bene Gesserit	5078872FY	Reverend Mother

Remember, you cannot see the S3 bucket in the AWS console, but here is the directory the table is located at.

Amazon S3 > Buckets > starburst101-handsonlab > w1-lester-martin-90210/ > myschema/ > dune_characters/

You can see the table name, `dune_characters`, as the deepest folder name which is nested below your schema name, `myschema`. Here is the sole file that is present in this folder.

Name	Type	Last modified	Size
20240125_043500_04345_fpnxe_3869b6f2-dee9-4bfa-aace-1a72eeee3edc	-	January 24, 2024, 23:35:02 (UTC-05:00)	1.2 KB

Note: The ORC file is not human-readable in a simple text editor, but the ORC tools jar is a relatively easy way to explore the contents of these files.

With managed tables, the SQL engine (Starburst Galaxy) is the framework that is creating these files. You do not have the opportunity for other systems/applications to create the files behind a managed table. This is part of the "management" that Starburst Galaxy is taking care of.

It is a common pattern to create an external table pointing to a data lake folder where files are being ingested. Then, at periodic intervals, this information is converted (via an `INSERT` statement) that selects the external table. This is a normal place to do transformations on the raw data's external table and then let the managed table that it is getting inserted into do the actual creation of the columnar files.

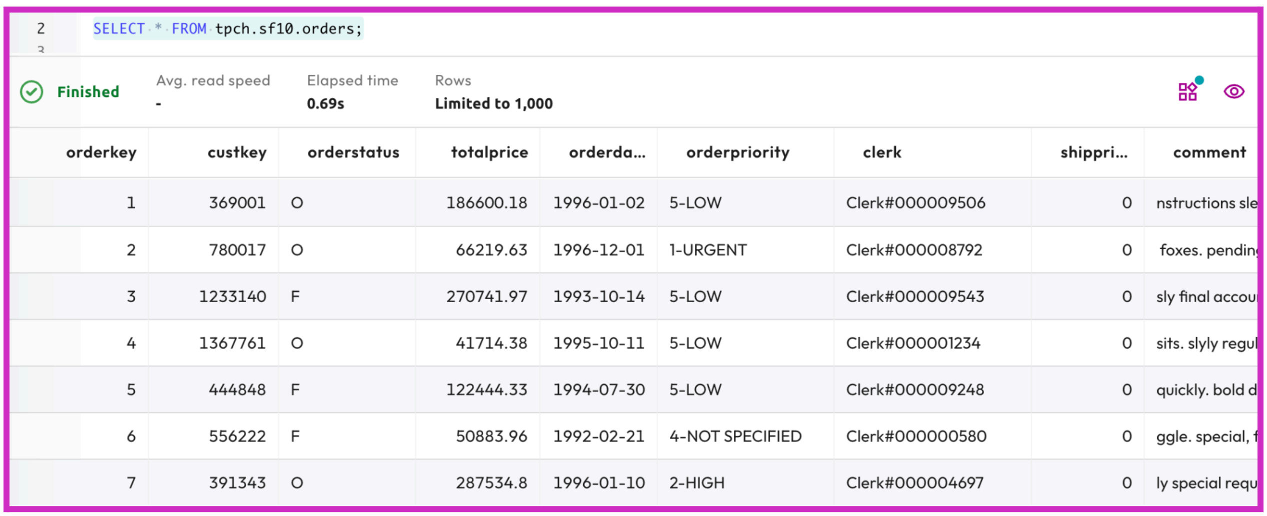
Even when the ingested data is of high quality, it is still a common practice to let this approach of loading it into a managed table. This allows the important task of the technical transformation to occur which itself enables higher performance tables.

Step 4 - Construct tables for performance tests

Starburst includes a deterministic data generator called the [TPCH connector](). For this exercise, you do not need to know much about this connector other than it is often used to populate other tables for benchmarking and performance comparisons.

Run this query to see what the `orders` table looks like.

```
SELECT * FROM tpch.sf10.orders;
```



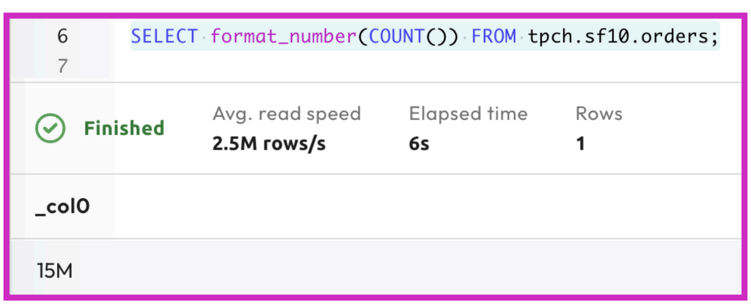
```
SELECT * FROM tpch.sf10.orders;
```

Finished | Avg. read speed - | Elapsed time 0.69s | Rows Limited to 1,000

orderkey	custkey	orderstatus	totalprice	orderda...	orderpriority	clerk	shippri...	comment
1	369001	O	186600.18	1996-01-02	5-LOW	Clerk#000009506	0	nstructions sle
2	780017	O	66219.63	1996-12-01	1-URGENT	Clerk#000008792	0	foxes. pendin
3	1233140	F	270741.97	1993-10-14	5-LOW	Clerk#000009543	0	sly final accou
4	1367761	O	41714.38	1995-10-11	5-LOW	Clerk#000001234	0	sits. slyly regul
5	444848	F	122444.33	1994-07-30	5-LOW	Clerk#000009248	0	quickly. bold d
6	556222	F	50883.96	1992-02-21	4-NOT SPECIFIED	Clerk#000000580	0	ggle. special, f
7	391343	O	287534.8	1996-01-10	2-HIGH	Clerk#000004697	0	ly special requ

Verify that this table has 15 million records in it by executing the following query.

```
SELECT format_number(COUNT()) FROM tpch.sf10.orders;
```



```
SELECT format_number(COUNT()) FROM tpch.sf10.orders;
```

Finished | Avg. read speed 2.5M rows/s | Elapsed time 6s | Rows 1

_col0
15M

Using the CTAS feature, create a managed table utilizing the JSON row-oriented file format and a second table leveraging the ORC columnar file format.

```
CREATE TABLE orders_json
WITH ( type = 'hive', format = 'json')
AS SELECT * FROM tpch.sf10.orders;

CREATE TABLE orders_orc
WITH ( type = 'hive', format = 'orc')
AS SELECT * FROM tpch.sf10.orders;
```

Notice it took about 90 seconds to create the JSON table and less than half that time to create the ORC table.

Run the following SQL to determine the number of files for each table along with the average file size and total amount of data for the whole table.

```
SELECT 'ORC' AS file_format, format_number(AVG("$file_size")) AS avg_file_size,
    COUNT() AS tot_nbr_files, format_number(SUM("$file_size")) AS tot_file_size
FROM
 (SELECT DISTINCT "$path", "$file_size" FROM orders_orc)
UNION
SELECT 'JSON' AS file_format, format_number(AVG("$file_size")) AS avg_file_size,
    COUNT() AS tot_nbr_files,format_number(SUM("$file_size")) AS tot_file_size
FROM
 (SELECT DISTINCT "$path", "$file_size" FROM orders_json);
```

file_format	avg_file_size	tot_nbr_files	tot_file_size
JSON	144M	4	575M
ORC	86.4M	4	346M

Step 5 - Compare performance for table row count

Run the following queries to determine the amount of time it takes to count the number of records from each table.

```
SELECT format_number(COUNT()) FROM orders_json;
SELECT format_number(COUNT()) FROM orders_orc;
```

Query	Elapsed time
SELECT format_nu...	9s
SELECT format_nu...	0.95s

Because the ORC file contains statistic, including file-level precalculations that include the total number of rows, it was an order of magnitude faster than the JSON table.

The JSON table had to have all columns read from every row of each file. That data was then discarded after it was read from the data lake as the engine simply updated counters until all rows were read and a final answer could be returned.

Step 6 - Compare performance with projection & filtering query

Run the following queries to determine the amount of time it takes to run a simple query with projection (limiting the columns to be returned) and filtering (adding a WHERE clause to limit the number of rows).

```sql
SELECT orderkey, totalprice FROM orders_json WHERE totalprice > 500000;
SELECT orderkey, totalprice FROM orders_orc WHERE totalprice > 500000;
```

Query	Elapsed time
SELECT orderkey, to...	40s
SELECT orderkey, to...	2s

The ORC table was 20x faster because the file had internal details of where possible matching records might be stored within the file itself. Additionally, only 2 of the 9 columns were read from the data lake. As before, all columns of all rows in all the files had to be read across the network for the JSON table

Step 7 - Compare performance while calculating aggregation values

Run the following queries to determine the amount of time it takes to run a classic `GROUP BY` aggregation query.

```sql
SELECT orderstatus, orderpriority, count() AS tot_nbr
 FROM orders_json
GROUP BY orderstatus, orderpriority
ORDER BY tot_nbr;

SELECT orderstatus, orderpriority, count() AS tot_nbr
 FROM orders_orc
GROUP BY orderstatus, orderpriority
ORDER BY tot_nbr;
```

Query	Elapsed time
SELECT orderstatus,...	42s
SELECT orderstatus,...	2s

The ORC table was 20x faster due to column statistics persisted in the files themselves. the file had internal details of where possible matching records might be stored within the file itself. Additionally, only 2 of the 9 columns were read from the data lake. As before, all columns of all rows in all the files had to be read across the network for the JSON table.

END OF LAB EXERCISE

Lab 4: Improve performance with partitioned tables (10 mins)

Learning objectives

- You will begin this lab by learning how table partitioning works and its logical benefits. You will create & load a partitioned table and then execute a single query that will showcase the primary goal of partitioning. You will finish the lab with a peek into more of the advanced reporting features of completed queries.

Prerequisites

- Lab 3 - Improve performance with columnar file formats

Activities

1. Understanding table partitioning
2. Create & populate a partitioned table
3. Compare query time
4. A deeper look at the performance improvement

Step 1 - Understanding table partitioning

In a nutshell, data lake table partitioning is a strategy where subfolders are created in a table's base folder. These tables are named with a pattern of `column_name=column_value`. The intention is to store all records that have a specific value for the column being partitioned into the same folder.

The following table presents an example; on the left is a description & the benefit and on the right is the data lake directory layout that could occur.

<table>
<tr>
<td>Assume table <code>hist_tbl</code> has 10 years of historical data.

Rather than dump all 10 years into a single folder, a folder could be created for each year (the example to the right only shows 3).

Then only data for the year identified in the folder name is placed within it.

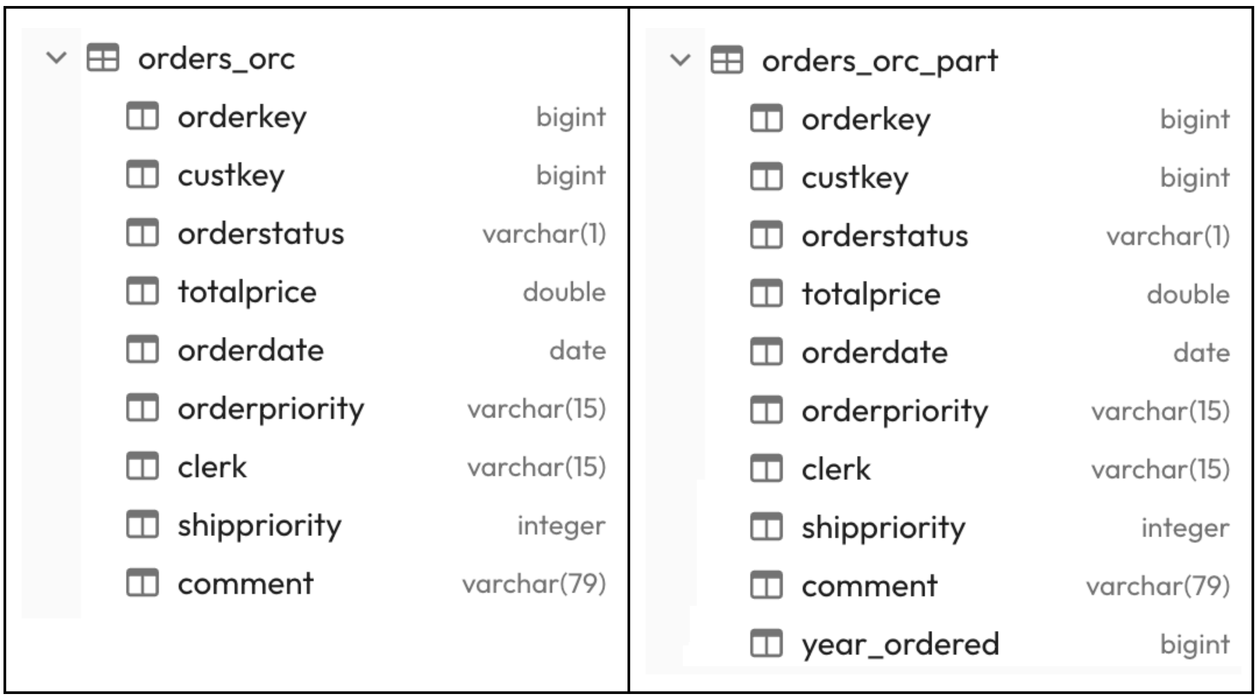
When a query includes the partitioned column in a <code>WHERE</code> clause, Starburst Galaxy can avoid reading partitions that do not match.</td>
<td><pre>.../cat/sch/hist_tbl
|-- year=2020
| |-- file1
| |-- file2
|-- year=2021
| |-- file3
| |-- file4
| |-- file5
|-- year=2022
| |-- file6
| |-- file7</pre></td>
</tr>
</table>

Step 2 - Create & populate a partitioned table

Run the following CTAS statement to create a partitioned table based on the `orders_orc` table previously created. Notice that a new field, `year_ordered`, was created and populated by extracting the year value from the `orderdate` column. Also notice, that this new column is identified in the `partition_by` property in the `WITH` clause.

```
CREATE TABLE orders_orc_part
WITH (type='hive', format='orc', partitioned_by=ARRAY['year_ordered'])
AS
SELECT *, extract(YEAR FROM orderdate) as year_ordered FROM orders_orc;
```

In the middle pane, expand `free-cluster`, then `webinar1`, and finally `myschema`. Ensure that you can see that `orders_orc_part` has an additional column named `year_ordered` as its final column that the `orders_orc` table does not have.



NOTE: The following are screenshots from S3 that you do not have direct access to.

Here is the base folder for the new table.

This folder contains subfolders for each of the unique values for the newly created `year_ordered` column.

Name	Type
year_ordered=1992/	Folder
year_ordered=1993/	Folder
year_ordered=1994/	Folder
year_ordered=1995/	Folder
year_ordered=1996/	Folder
year_ordered=1997/	Folder
year_ordered=1998/	Folder

Drilling into the `year_ordered=1993` folder you can see files that only contain data with rows where the `orderdate` values are within that year.

Name	Type	Last modified	Size
20240125_074148_07543_fpnxe_316208b3-c1a9-4ba7-b388-7f3ab20dfd2c	-	January 25, 2024, 02:42:42 (UTC-05:00)	36.9 MB
20240125_074148_07543_fpnxe_5e71c060-e2e0-4702-a70e-7f0fff99aee0	-	January 25, 2024, 02:42:39 (UTC-05:00)	14.9 MB

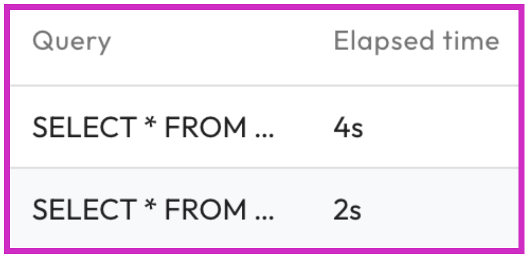
Remember, you do not have the privileges needed to view this in S3 directly. The last few screenshots were included to help you understand what is happening on the data lake.

Step 3 - Compare query time

Run the following two SQL statements. Notice that for the partitioning logic to work, the query writer must utilize the new partitioned column in their `WHERE` clause.

```
SELECT * FROM orders_orc
WHERE totalprice > 500000
  AND extract(YEAR FROM orderdate) = 1997;

SELECT * FROM orders_orc_part
WHERE totalprice > 500000
  AND year_ordered = 1997;
```

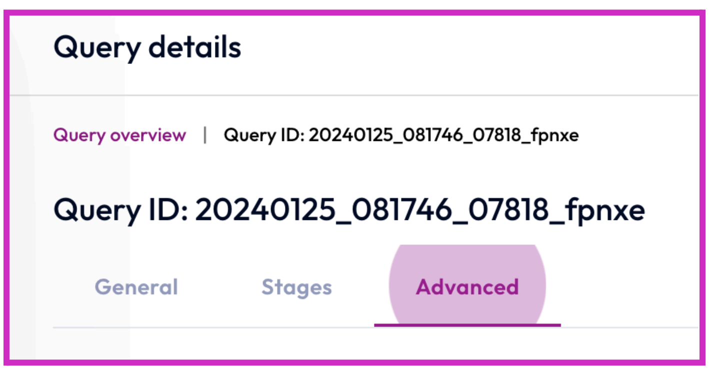



The 2x improvement of elapsed time for this relatively small test is respective and comes about because the query engine only had to look into a single folder for the partitioned table.

Step 4 - A deeper look at the performance improvement

For each of the two query executions, click on the **See query details** “eyeball” icon on the results banner above the query results.

On the **Query details** page that renders, select the **Advanced** tab.



Locate the **Execution details** panel. Your results should be similar to these.

`orders_orc`	`order_orc_part`
Execution details	**Execution details**
ELAPSED TIME **4.23s**	ELAPSED TIME **2.21s**
CPU TIME **1.48s**	CPU TIME **0.32s**
PARALLELISM **0.35**	PARALLELISM **0.146**
ACTIVE **12%**	ACTIVE **13%**
ROWS READ **1.7M**	ROWS READ **244K**
BYTES READ **265 MB**	BYTES READ **48.2 MB**

You can see the 2x improvement to **ELAPSED TIME** that was previously discussed, but notice that **CPU TIME** was > 4x better for the partitioned table. This gap will widen dramatically when you have significantly more data present and the engine can avoid accessing a high percentage of the partitioned folders.

Despite these tables not being large at all, the much smaller values for **ROWS READ** and **BYTES READ** are due to the engine being able to prune the partitions and only read data that could be present in the final results.

END OF LAB EXERCISE

Lab 5: Exploring federated queries (5 mins)

Learning objectives

- This lab will showcase how federation works and what it allows you to do. You will validate a join across multiple catalogs functions as expected.

Prerequisites

- Lab 3 - Improve performance with columnar file formats

Activities

1. Understand query federation
2. Identify the tables to join
3. Create the base join statement
4. Refine the federated query

Step 1 - Understand query federation

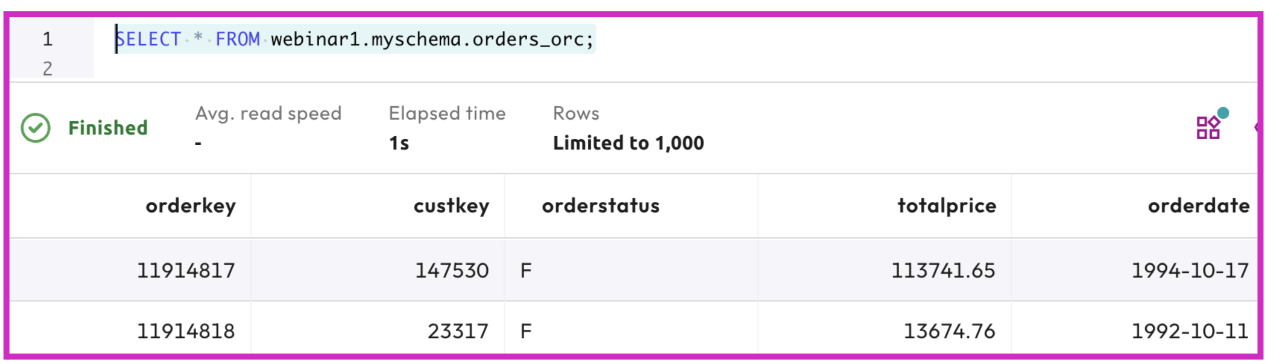
Starburst allows you a single point of access to a variety of data data sources. The list of connectors shows what is available. Additionally, having connections to multiple data sources also allows for query federation.

Simply put, you can access data from multiple systems within a single SQL query. For example, you could join historical log data stored in an S3 object storage bucket with customer data stored in an Oracle database.

Step 2 - Identify the tables to join

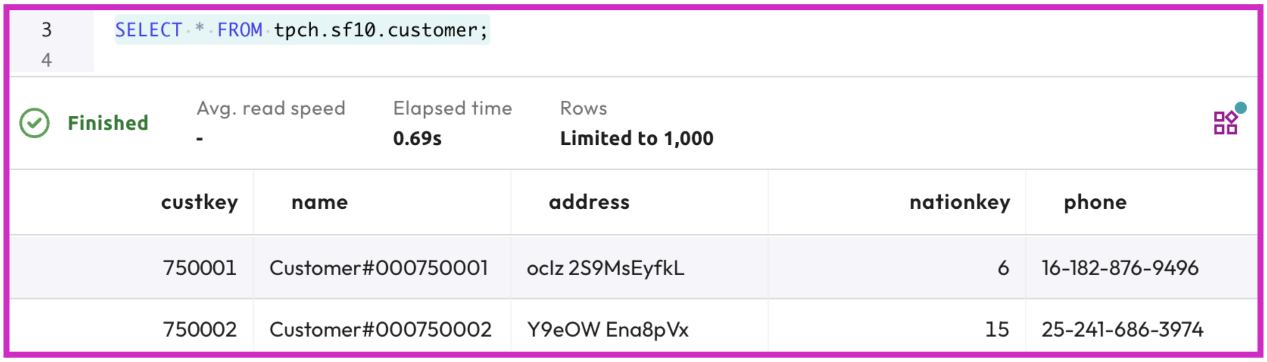
Run the following query and notice that the results include a `custkey` field.

```
SELECT * FROM webinar1.myschema.orders_orc;
```



Run the following query on the table you need to join on and notice the join column has the same name; `custkey`.

```
SELECT * FROM tpch.sf10.customer;
```



Step 3 - Create the base join statement

Run the following query to ensure the federated join across two different catalogs, each using different connector types, will fundamentally work.

```
SELECT *
  FROM webinar1.myschema.orders_orc AS o
  JOIN tpch.sf10.customer AS c
    ON (o.custkey = c.custkey);
```

Step 4 - Refine the federated query

Run the following enhanced federated query.

```
SELECT c.mktsegment, c.nationkey, o.orderstatus,
       format('%,.2f', AVG(totalprice)) AS avg_totalprice
  FROM webinar1.myschema.orders_orc AS o
  JOIN tpch.sf10.customer AS c
    ON (o.custkey = c.custkey)
 WHERE o.orderstatus = 'O'
   AND c.nationkey IN (8, 9)
 GROUP BY c.mktsegment, c.nationkey, o.orderstatus
 ORDER BY c.nationkey, c.mktsegment;
```

Finished | Avg. read speed **2.8M rows/s** | Elapsed time **5s** | Rows **10**

mktsegment	nationkey	orderstatus	avg_totalprice
AUTOMOBILE	8	O	150,187.34
BUILDING	8	O	149,751.37
FURNITURE	8	O	150,230.02

END OF LAB EXERCISE