



Galaxy and Iceberg Introduction Workshop

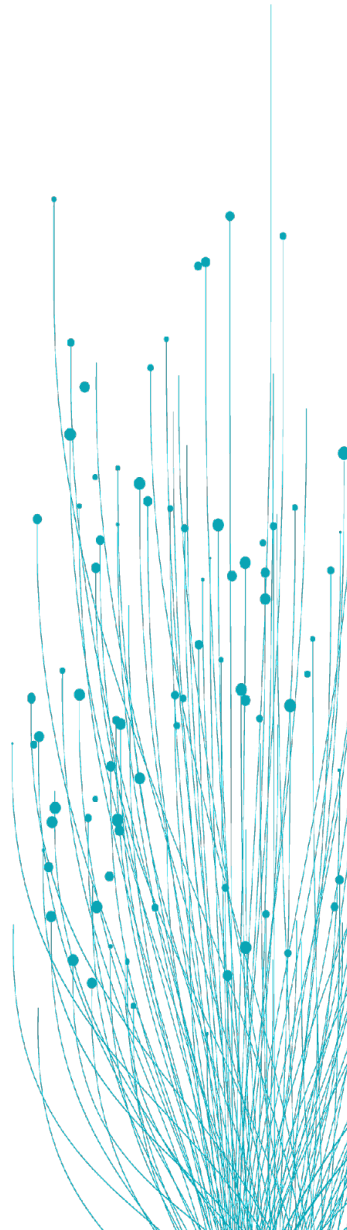
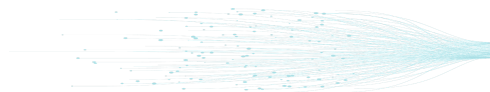


TABLE OF CONTENTS

1. Setup a Galaxy Trial	2
2. Create a New Cluster	2
3. Create an Iceberg Catalog	5
4. Querying data in Starburst Galaxy	8
5. Importing Sample Data	11
6. Querying the Sample Data in Iceberg Tables	13
7. Data Modification Language (DML) with Iceberg	14
8. Merge Statement with Iceberg	16
9. Time travel	19
10. Working with Partitions	23
11. Altering Table Metadata	28
Appendix A	30
Links to Starburst Online Documentation	30
Appendix B	31
Ingesting raw csv data	31
Appendix C	34
Tips and Tricks	34
Appendix D	36
SQL in full	36



1. Setup a Galaxy Trial

<https://www.starburst.io/platform/starburst-galaxy>

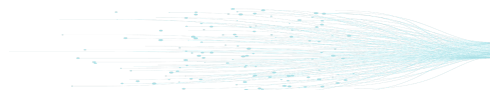
2. Create a New Cluster

Select the 'Create Cluster' button having selected 'Clusters' on the left-hand menu. Create the Cluster in the 'Europe (Ireland)' region. Give the Cluster a name, e.g., 'tis2023'.

Select the Standard Cluster Type, and the Free-Tier Size. If you have spare credits, you can select a specific Cluster Size (e.g., X-Small, Small, etc.).

The 'Free' Cluster is adequate for the workshop.

Select the 'Europe (London)' region.



Create a new cluster ✕

Cluster name *

Must start with a letter and only use lowercase letters (a-z), numbers (0-9), and hyphens (-)

Catalogs

Cloud provider region *

Cluster type

Execution mode *

Cluster size *

Idle shutdown time

The maximum idle time before a cluster is automatically suspended.

Advanced settings ▼

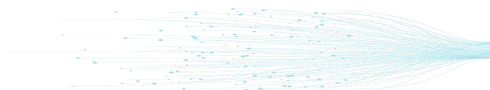
Note: Ensure the tpch and tpcds catalogs are added to the cluster.

Also, setting the 'Idle shutdown time' to a value greater than the default is recommended for this workshop (e.g., set to 30 minutes). Given the duration of the workshop, it is preferable that time spent pausing and re-started clusters is minimized.

Under 'Access control' on the left-hand side of the screen – select 'Roles and privileges'.

Drill into the 'accountadmin' role (assuming this is your user Role – shown on the top right of the screen).

Under the 'Privileges' tab, add on a Location privilege as per the below:



Add privilege

Assign to: accountadmin

Allow this role to access all or specific catalogs, tables, or clusters within your organization. Refer to our detailed [documentation on assigning privileges to roles](#) .

What would you like to modify privileges for?

- Account Catalog Cluster Schema Table Column Location
- Function

Enter a storage location.

A storage location starts with **s3://**, **gs://**, or **abfs://** and ends with **/***

Storage location name *

s3://galaxy-data-innovation/*

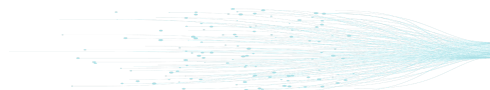
Create SQL

Cancel

Add privileges

Ensure the 'Create SQL' box is checked.

Click on the 'Add Privilege' button.



3. Create an Iceberg Catalog

Select 'Catalogs' on the left-hand, click the 'Create Catalog' button.
Select s3 as the Data Source.

Complete the wizard. Input the name of the catalog, e.g., iceberg (lower case).

Input the AWS Access Key and Secret:

Access Key:	Provided on the day.
Secret:	Provided on the day.
Default s3 bucket:	galaxy-data-inovation
Directory Name:	iceberg

Leave the Metastore as Starburst Galaxy.
Turn on the buttons to allow creating External Tables and Writes.

Select 'Iceberg' as the default Table format. Test the Connection.

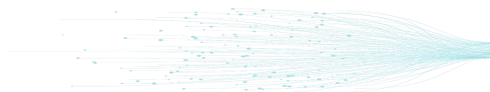
If all is ok – Click on Connect Catalog.

The next screen relates to Access Controls.

Click 'Save Access Controls'.

In the 'Add to Cluster' Screen – add the Iceberg catalog to the Cluster you created previously.

You will now be taken to the Query Editor to query data.



Create catalog

- ✓ Select a data source
- 2 Configure the connection
- 3 Set permissions (optional)
- 4 Add to cluster (optional)

Amazon S3

Configure your catalog to query objects in Amazon S3. Learn more about [connecting to S3](#).

Name and description

Provide a unique name to identify the catalog in your SQL queries in the query editor and other client tools. The namespace for a table is typically <catalog_name>.<schema_name>.<table_name>

Catalog name *

 ?
Must start with a letter and only use lowercase letters (a-z), numbers (0-9), and underscores (_).
 Description
 ?

Authentication to S3

Choose the [authentication mechanism](#) to connect to S3.

Authentication with *

- Cross account IAM role AWS access key

AWS access key for S3 *

 ?
 AWS secret key for S3 *
 ?

Metastore configuration

Configure access to the metastore to provide metadata and mapping information about the objects stored in Amazon S3.

Metastore type *

- AWS Glue Hive Metastore Starburst Galaxy

Default S3 bucket name *

 ?
 Default directory name *
 ?
 Allow creating external tables ?
 Allow writing to external tables ?

Default table format

Select the default table format used for creating new tables. The catalog will be able to read from any type. [Check out our docs](#) to learn more.

Default table format *

- Iceberg Hive Delta Lake

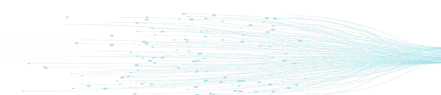
Test connection

Validate that the network configuration allows Starburst Galaxy to connect to the data source.

[Test connection](#)

[← Back](#)

[Connect catalog](#)

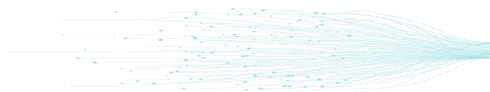


Ensure you have the right Privilege to work with Objects in the Catalog.

Select the 'Access Control' tab on the left-hand side of the screen.

Drill into the 'accountadmin' role. Go to 'Privileges' tab.

View the 'Privileges' assigned. The Catalog you created should be listed under 'Catalogs'.



4. Querying data in Starburst Galaxy

We will use the Sample tpch dataset to run some queries. Ensure the Cluster you created has the tpch catalog attached. Start the Cluster. Cut and paste the following SQL into the Query Editor and run:

```
SELECT
  COUNT(*) AS LINEITEMS,
  Q.PARTKEY,
  R.NAME,
  P.ORDERKEY
FROM
  tpch.tiny.customer AS R
  INNER JOIN tpch.tiny.orders AS P ON R.CUSTKEY = P.CUSTKEY
  INNER JOIN tpch.tiny.lineitem AS Q ON P.ORDERKEY = Q.ORDERKEY
GROUP BY
  Q.PARTKEY,
  R.NAME,
  P.ORDERKEY
HAVING
  COUNT(*) > 1
ORDER BY
  1 DESC;
```

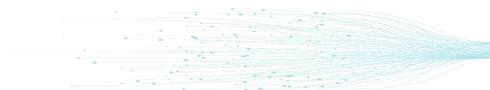
In the SQL, we are using the 'tiny' schema. You can use different scale factor schemas, e.g., 'sf1', 'sf100', etc.

Note: the 'sf1' schema contains ~ 1.2GB of data, the 'sf100' schema contains ~ 120GB of data.

The tpch and tpchds data is generated on the fly, so some compute is used in the generation of the data.

For an example of a multi-way Table join:

```
SELECT
  COUNT(*) AS LINEITEMS,
  Q.PARTKEY,
  R.NAME,
  P.ORDERKEY,
  T.SUPPKEY,
```



```

U.PARTKEY,
V.NAME,
W.NAME as NATION,
X.NAME as REGION
FROM
tpch.tiny.customer AS R
INNER JOIN tpch.tiny.orders AS P ON R.CUSTKEY = P.CUSTKEY
INNER JOIN tpch.tiny.lineitem AS Q ON P.ORDERKEY = Q.ORDERKEY
INNER JOIN tpch.tiny.supplier AS T ON Q.SUPPKEY = T.SUPPKEY
INNER JOIN tpch.tiny.partsupp AS U ON T.SUPPKEY = U.SUPPKEY
INNER JOIN tpch.tiny.part AS V ON U.PARTKEY = V.PARTKEY
INNER JOIN tpch.tiny.nation AS W ON T.NATIONKEY = W.NATIONKEY
INNER JOIN tpch.tiny.region AS X ON W.REGIONKEY = X.REGIONKEY
WHERE
X.NAME = 'EUROPE'
GROUP BY
Q.PARTKEY,
R.NAME,
P.ORDERKEY,
T.SUPPKEY,
U.PARTKEY,
V.NAME,
W.NAME,
X.NAME
HAVING
COUNT(*) > 1
ORDER BY
1 DESC;

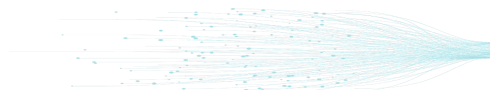
```

Note: If you want to test Query Federation – simply change the schema for certain tables to use the ‘sf1’ schema, and others to use the ‘tiny’ schema, for example:

```

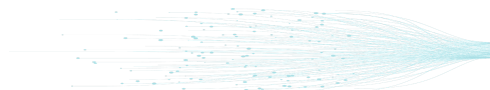
SELECT
COUNT(*) AS LINEITEMS,
Q.PARTKEY,
R.NAME,
P.ORDERKEY
FROM
tpch.sf1.customer AS R
INNER JOIN tpch.sf1.orders AS P ON R.CUSTKEY = P.CUSTKEY

```



```
INNER JOIN tpch.tiny.lineitem AS Q ON P.ORDERKEY = Q.ORDERKEY
GROUP BY
  Q.PARTKEY,
  R.NAME,
  P.ORDERKEY
HAVING
  COUNT(*) > 1
ORDER BY
  1 DESC;
```

Note: the above is joining tables between two schemas in the same catalog – so not true Query Federation. However – if you have matching keys in tables that are defined in Galaxy on different sources – then you can join as above. This assumes you have access to those Tables.



5. Importing Sample Data

We have Online Retail Transaction Data to use as the sample data. We are using data in a modern file format – Parquet in this case. Galaxy supports Iceberg tables in Parquet and ORC file format.

First create a Schema under the Iceberg Catalog you created previously:

```
CREATE SCHEMA iceberg.student1
WITH
  (LOCATION = 's3://galaxy-data-inovation/iceberg/student1/');
```

We then register an Iceberg Table in Galaxy to read in the Parquet files and Metadata:

```
CALL iceberg.system.register_table (
  schema_name => 'student1',
  table_name => 'sales_land',
  table_location => 's3://galaxy-data-inovation/iceberg/student1/'
);
```

Note: In this Lab we are assuming the Data Files have been processed and have been made available in Iceberg format.

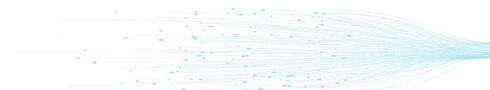
We are making this assumption in the interest of time. In many cases the Source data will not be in a modern table format. Common examples would be files in text or csv format.

There is a Section in the Appendix which walks through the additional steps to ingest from csv files.

Validate the Table has been registered, run the following:

```
ANALYZE iceberg.student1.sales_land;
SHOW STATS FOR iceberg.student1.sales_land;
SELECT COUNT(*) from iceberg.student1.sales_land;
```

We will create another Table – called ‘sales1’, to highlight how a Partition is created. In a later Section, we will add a Partition to an existing table which already contains data. The Section will also review if the column – ‘country’ is a good candidate to use for partitioning.

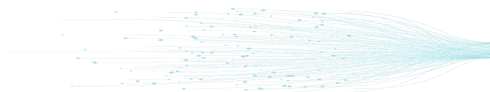


Note: the below will only work if you created the 'sales_land' table.

```
CREATE TABLE
  iceberg.student1.sales1
WITH
  (
    FORMAT = 'PARQUET',
    format_version = 2,
    partitioning = ARRAY['country'],
    type = 'ICEBERG'
  ) AS
SELECT
  *
FROM
  iceberg.student1.sales_land
```

Validate and profile the Table:

```
ANALYZE iceberg.student1.sales1;
SHOW STATS FOR iceberg.student1.sales1;
SELECT COUNT(*) from iceberg.student1.sales1;
```



6. Querying the Sample Data in Iceberg Tables

Let's run some Queries on the Sample Data to get an understanding of the Data Profile:

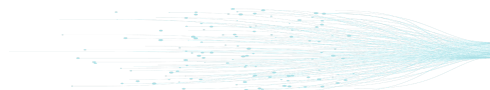
```
SELECT
  COUNT(*) count,
  COUNTRY
from
  iceberg.student1.sales1
GROUP BY
  COUNTRY
ORDER BY
  1 DESC;
```

```
SELECT * FROM "iceberg"."student1"."sales" LIMIT 10;
```

Iceberg Tables contain additional metadata columns. We will work with the partitioning metadata in a later section.

To view this metadata, run the SQL below (or change the catalog, schema, table name if required):

```
SELECT * FROM "iceberg"."student1"."sales1" LIMIT 10;
SELECT * FROM "iceberg"."student1"."sales1$properties" LIMIT 10;
SELECT * FROM "iceberg"."student1"."sales1$history" LIMIT 10;
SELECT * FROM "iceberg"."student1"."sales1$snapshots" LIMIT 10;
SELECT * FROM "iceberg"."student1"."sales1$manifests" LIMIT 10;
SELECT * FROM "iceberg"."student1"."sales1$partitions" LIMIT 10;
SELECT * FROM "iceberg"."student1"."sales1$files" LIMIT 10;
SELECT *, "$path", "$file_modified_time" FROM "iceberg"."student1"."sales1" ;
```



7. Data Modification Language (DML) with Iceberg

Let's Insert some records into the Table:

```
INSERT INTO
  "iceberg"."student1"."sales1"
VALUES
  ('999999', '88888', 'Galaxy T-Shirt', 1, DATE('2023-05-11'), 10.2, 777, 'Sweden'),
  ('555555', '44444', 'Cmd BunBun', 1, DATE('2023-05-12'), 20.20, 333, 'Sweden');
```

The above is an example of an Insert statement. For bulk imports, often the CTAS (CREATE TABLE AS SELECT) syntax is used, to load in a Table to a new Table based on the SELECT clause).

We will now run Updates on a particular value in a Table. First let's look at a row:

```
SELECT
*
FROM
  "iceberg"."student1"."sales1"
WHERE
  invoice = '494234';
```

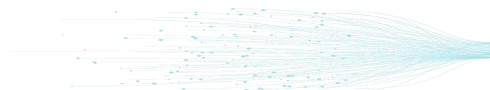
The value of the 'price' column for this item is '5.00'. Let's update the price by 10% using the following SQL:

```
UPDATE "iceberg"."student1"."sales1"
SET
  price = price * 1.1
where
  invoice = '494234';
```

The same SELECT statement against the row should show the updated value for price. This may seem like a trivial capability to an experienced DBAs – but basic DML tasks were often constrained or not available on legacy data lake Table Formats. This is no longer the case with Iceberg.

The above logic would work if there were > 1 row with the invoice number '494234' (e.g., try with invoice '537434').

Deleting a row is straight-forward, for example:

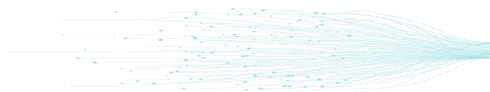


```
DELETE FROM "iceberg"."student1"."sales1"  
WHERE  
    invoice = '494234'
```

Validate the row has been deleted:

```
SELECT  
    *  
FROM  
    "iceberg"."student1"."sales1"  
WHERE  
    invoice = '494234'
```

In this case one row, but multiple rows could have been deleted if the WHERE clause matched multiple rows.



8. Merge Statement with Iceberg

When ingesting and processing data, inserting records as-is into a Table is a common pattern. This is straight forward to handle as each record is treated as being independent of each other. An example of this would be loading stock ticker records. Each record is loaded, as each record or event is discrete – even though there will be records for the same Stock (e.g. Acme Corp), any aggregations are often done as a separate process.

Another common pattern is where there is a need for logic to determine if a record already exists – and if so, what Data Manipulations to perform.

This is where the MERGE statement is often used. The MERGE contains logic to evaluate the data – and to determine if a row should be UPDATED, DELETED, or INSERTED.

One MERGE statement can often replace multiple SQL statements achieving the same function.

In the Workshop, we will create the new data table:

CREATE TABLE

```
iceberg.student1.sales2
(
  invoice VARCHAR,
  stockcode VARCHAR,
  description VARCHAR,
  quantity INT,
  invoicedate date,
  price decimal (8, 2),
  customerid INT,
  country varchar
)
```

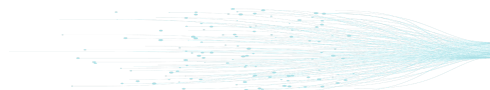
We will input two rows into this Table.

INSERT INTO

```
"iceberg"."student1"."sales2"
```

VALUES

```
('123456', '98765', 'Iceberg Badge', 1, DATE('2023-05-11'), 10.2, 222, 'Sweden'),
('555555', '44444', 'Starburst Swag', 1, DATE('2023-05-12'), 20.20, 333, 'Sweden')
```



To best understand what the MERGE is doing, let's join the new and existing data (or source and target) tables, and view the Matching records – which will get processed.

SELECT

```
a.*,  
b.*
```

FROM

```
"iceberg"."student1"."sales1" a
```

INNER JOIN

```
"iceberg"."student1"."sales2" b on a.customerid = b.customerid
```

The above (on the first time you run, and before the MERGE), should return 1 x row (assuming you ran the INSERT statement above).

invoice	stockcode	description	quantity	invoicedate	price	customerid	country
555555	444444	Cmd BunBun	1	2023-05-12	20.20	333	Sweden

This is informing us that the Merge statement will find one Matching row. Looking at the new data Table – we can see there are two rows that will be processed by the MERGE.

invoice	stockcode	description	quantity	invoicedate	price	customerid	country
123456	98765	Iceberg Badge	1	2023-05-11	10.20	222	Sweden
555555	444444	Starburst Swag	1	2023-05-12	20.20	333	Sweden

The MERGE will UPDATE the Matching row with a new 'description' – 'Cmd BunBun' will get replaced with 'Starburst Swag'.

The second row was not matched so is treated as an INSERT.

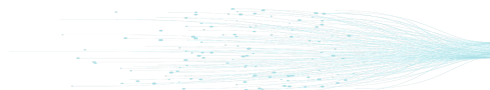
Run the MERGE to confirm this.

```
MERGE INTO "iceberg"."student1"."sales1" AS a USING "iceberg"."student1"."sales2" AS b  
ON (a.customerid = b.customerid) WHEN MATCHED
```

```
and a.description != b.description THEN
```

```
UPDATE
```

```
SET
```



```

description = b.description WHEN NOT MATCHED THEN INSERT (
  invoice,
  stockcode,
  description,
  quantity,
  invoicedate,
  price,
  customerid,
  country
)
VALUES
(
  b.invoice,
  b.stockcode,
  b.description,
  b.quantity,
  b.invoicedate,
  b.price,
  b.customerid,
  b.country
);

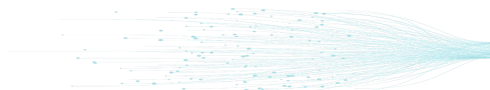
```

Post the MERGE statement, the Inner Join of the two tables should now look like:

invoice	stockcode	description	quantity	invoicedate	price	customerid	country
555555	444444	Starburst Swag	1	2023-05-12	20.20	333	Sweden
123456	98765	Iceberg Badge	1	2023-05-11	10.20	222	Sweden

The above confirms the Merge:

- Updated the Matching row with a new 'description'.
- Inserted the row that did not match.



9. Time travel

Time travel enables Queries to view data at a previous snapshot in time.

It can be used to assist:

- Discovery of historic data that has since been made unavailable.
- Restore Data that has been deleted or modified.
- Provides a reassurance that data can be recovered.

This exercise will require the student to input some of the values into SQL statements. This because in Time travel – some of the scenarios will be specific to your own Iceberg Tables.

Note: It is useful to be aware of some of the Galaxy date/time functions, for example try running the below – in one block (select all lines and execute):

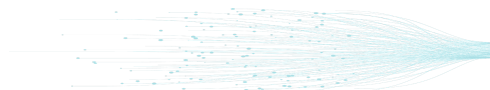
```
SELECT current_date;
SELECT current_timestamp; -- same as now()
SELECT now();           -- same as current_timestamp
SELECT current_time;
SELECT localtimestamp;
SELECT localtime;
SELECT current_time + interval '1' minute ;
SELECT current_time - interval '2' hour;
SELECT current_date + interval '3' day ;
SELECT current_date - interval '4' month ;
SELECT current_date + interval '5' year ;
SELECT current_timestamp + interval '30' second ;
VALUES now();
```

Let us look at the snapshots for the "iceberg"."student1"."sales1" Table:

```
SELECT * FROM "iceberg"."student1"."sales1$snapshots" LIMIT 10;
```

Let's perform an Insert into the sales1 table, notice one column is using the 'current date' value:

```
INSERT INTO
  "iceberg"."student1"."sales1"
```

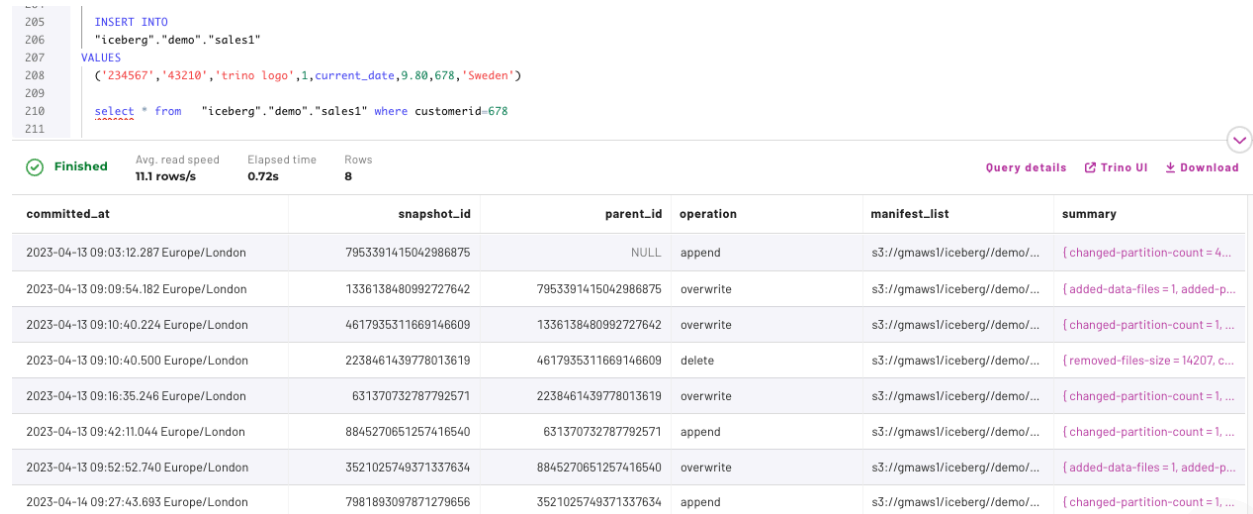


VALUES

```
('234567', '43210', 'trino logo', 1, current_date, 9.80, 678, 'Sweden');
```

Again, run the snapshot command:

```
SELECT * FROM "iceberg"."student1"."sales1$snapshots" LIMIT 10;
```



The screenshot shows a Trino query execution interface. The query executed was:

```
INSERT INTO
  "iceberg"."demo"."sales1"
VALUES
  ('234567', '43210', 'trino logo', 1, current_date, 9.80, 678, 'Sweden')
select * from "iceberg"."demo"."sales1" where customerid=678
```

The execution status is **Finished** with an average read speed of 11.1 rows/s, an elapsed time of 0.72s, and 8 rows returned. The interface also provides links for [Query details](#), [Trino UI](#), and [Download](#).

committed_at	snapshot_id	parent_id	operation	manifest_list	summary
2023-04-13 09:03:12.287 Europe/London	7953391415042986875	NULL	append	s3://gmaws1/iceberg/demo/...	[changed-partition-count = 4, ...
2023-04-13 09:09:54.182 Europe/London	1336138480992727642	7953391415042986875	overwrite	s3://gmaws1/iceberg/demo/...	[added-data-files = 1, added-p...
2023-04-13 09:10:40.224 Europe/London	4617935311669146609	1336138480992727642	overwrite	s3://gmaws1/iceberg/demo/...	[changed-partition-count = 1, ...
2023-04-13 09:10:40.500 Europe/London	2238461439778013619	4617935311669146609	delete	s3://gmaws1/iceberg/demo/...	[removed-files-size = 14207, c...
2023-04-13 09:16:35.246 Europe/London	631370732787792571	2238461439778013619	overwrite	s3://gmaws1/iceberg/demo/...	[changed-partition-count = 1, ...
2023-04-13 09:42:11.044 Europe/London	8845270651257416540	631370732787792571	append	s3://gmaws1/iceberg/demo/...	[changed-partition-count = 1, ...
2023-04-13 09:52:52.740 Europe/London	3521025749371337634	8845270651257416540	overwrite	s3://gmaws1/iceberg/demo/...	[added-data-files = 1, added-p...
2023-04-14 09:27:43.693 Europe/London	7981893097871279656	3521025749371337634	append	s3://gmaws1/iceberg/demo/...	[changed-partition-count = 1, ...

There should be > 1 snapshot file present (this depends on your activity in the lab).

If you cut and paste the 'snapshot_id' to a version before you ran the INSERT (with the 'description' = 'trino logo'), then you should see no records.

The value of the 'snapshot_id' will be different for each student – as it depends on the value generated by Iceberg in your lab.

In this scenario, from the above screen shot, the snapshot_id just before the INSERT is **3521025749371337634**

SELECT

*

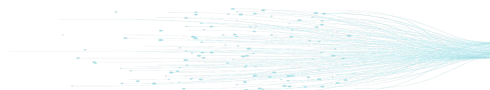
FROM

```
"iceberg"."student1"."sales1" FOR VERSION AS OF 3521025749371337634
```

where

```
customerid = 678
```

The above returns no row – as the INSERT had not taken place.



The same statement – without reference to the snapshot should show the row:

```
SELECT
  *
FROM
  "iceberg"."student1"."sales1"
WHERE
  customerid = 678
```

You can also use a timestamp to run time travel queries. The timestamp can be a literal or else you can use a system date/time function.

An example of using a system function for the time travel query:

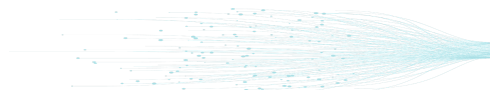
```
SELECT
  *
FROM
  "iceberg"."student1"."sales1" FOR TIMESTAMP AS OF (current_timestamp - interval '30'
minute)
WHERE
  customerid = 678;
```

Adjust the interval (i.e. the value of '30' above) – to before or after the point-in-time where you inserted the row with 'customerid' = 678.

The Query Result should change - if your date/time is before or after the INSERT.

The timestamp can be in a literal form (please adjust the value to match your present date/time):

```
SELECT
  *
FROM
  "iceberg"."student1"."sales1" FOR TIMESTAMP AS OF TIMESTAMP '2023-04-14 11:00:29.803
Europe/Vienna'
WHERE
  customerid = 678;
```



Adjust the date/time – to before or after the point-in-time where you inserted the row with 'customerid' = 678. The Query Results will change either side of this time boundary.

In the event you were satisfied there was an error in updating data in an Iceberg Table, you can revert to a previous snapshot.

In this case we will assume we don't want to have the recent INSERT row in the Table.

Again – check the snapshot versions you have available:

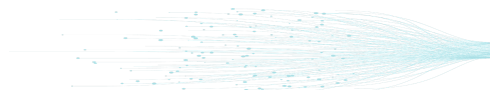
```
SELECT * FROM "iceberg"."student1"."sales1$snapshots" LIMIT 20;
```

And select the snapshot id with the version of the Table at the point in time before the INSERT.

In this case it is - 3521025749371337634

```
CALL iceberg.system.rollback_to_snapshot('student1', 'sales1', 3521025749371337634)
```

After the above statement is run – the Table no longer contains the row from the previous INSERT.



10. Working with Partitions

We previously created the "iceberg"."student1"."sales1" partitioned on 'country':

```
partitioning = ARRAY['country']
```

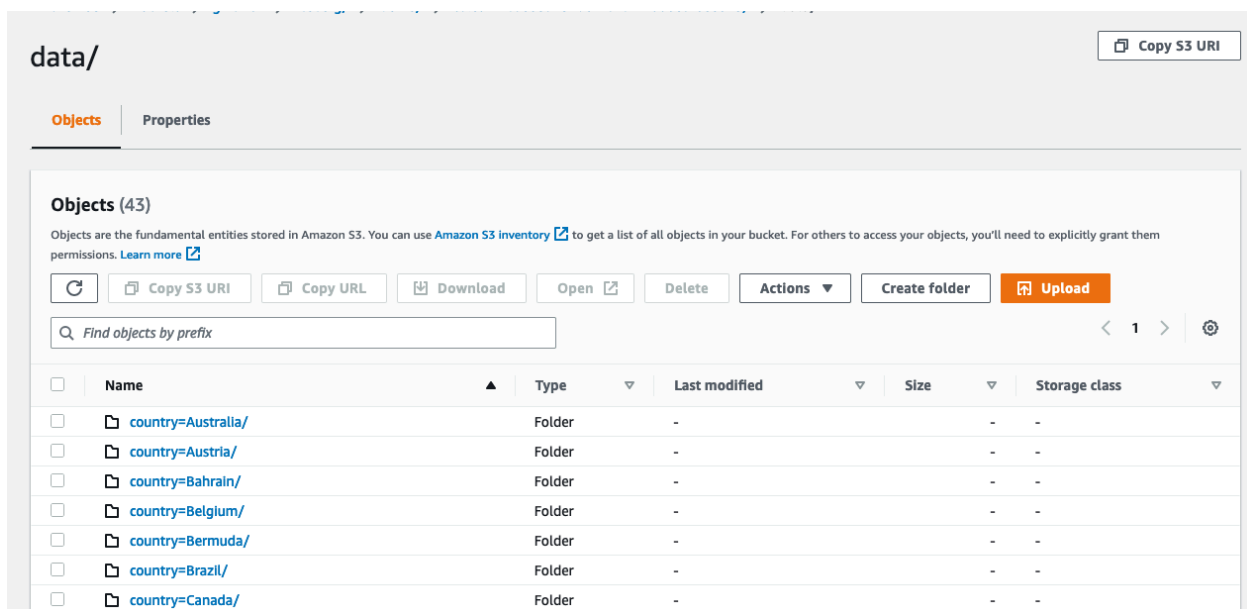
To view the Partitions on the Table, you can run the command:

```
SELECT * FROM "iceberg"."student1"."sales1$partitions" LIMIT 50;
```

View the records, files, and data values in each partition.

Note: This section about Object Store is for information only. For many users, viewing or interacting with the Object Store (buckets, folders, files, formats, etc.) is not something you would be permitted or required to do (but DevOps, SecOps, Cloud Admins, Data Engineers, Data Managers, Data Scientists, etc., would often be required to access).

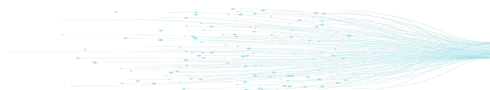
In the Object Store (s3), each Partition resides in a Folder:



The screenshot shows the Amazon S3 console interface for a bucket named 'data/'. The 'Objects' tab is selected, showing a list of 43 objects. The objects are organized into folders by country. The table below represents the data shown in the screenshot:

Name	Type	Last modified	Size	Storage class
country=Australia/	Folder	-	-	-
country=Austria/	Folder	-	-	-
country=Bahrain/	Folder	-	-	-
country=Belgium/	Folder	-	-	-
country=Bermuda/	Folder	-	-	-
country=Brazil/	Folder	-	-	-
country=Canada/	Folder	-	-	-

Under each folder you will find the Data Files (we defined them as Parquet file format) in the Iceberg Table definition.



Objects (4)

Objects are the fundamental entities stored in Amazon S3. You can use [Amazon S3 inventory](#) to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. [Learn more](#)

Find objects by prefix

<input type="checkbox"/>	Name	Type	Last modified	Size
<input type="checkbox"/>	20230413_080253_03723_bk5de-361971d6-21ad-4275-8155-0cefd8918b44.parquet#%2Ficeberg%2F%2Fdemo%2Fsales1-1bae03a232d04fc7b4f28806af0ee045%2Fdata%2Fcountry%3DAustralia%2F20230413_080253_03723_bk5de-361971d6-21ad-4275-8155-0cefd8918b44.parquet	parquet	April 13, 2023, 09:03:09 (UTC+01:00)	7.0 MB
<input type="checkbox"/>	20230413_080253_03723_bk5de-8235cdd6-0deb-47ef-97be-8dceb5232ea7.parquet#%2Ficeberg%2F%2Fdemo%2Fsales1-1bae03a232d04fc7b4f28806af0ee045%2Fdata%2Fcountry%3DAustralia%2F20230413_080253_03723_bk5de-8235cdd6-0deb-47ef-97be-8dceb5232ea7.parquet	parquet	April 13, 2023, 09:03:10 (UTC+01:00)	7.0 MB
<input type="checkbox"/>	20230413_080253_03723_bk5de-8368af93-ceb0-40a6-a44d-c96a578566f8.parquet#%2Ficeberg%2F%2Fdemo%2Fsales1-1bae03a232d04fc7b4f28806af0ee045%2Fdata%2Fcountry%3DAustralia%2F20230413_080253_03723_bk5de-8368af93-ceb0-40a6-a44d-c96a578566f8.parquet	parquet	April 13, 2023, 09:03:08 (UTC+01:00)	9.0 MB
<input type="checkbox"/>	20230413_080253_03723_bk5de-8a431de6-d810-4386-8411-796352eda108.parquet#%2Ficeberg%2F%2Fdemo%2Fsales1-1bae03a232d04fc7b4f28806af0ee045%2Fdata%2Fcountry%3DAustralia%2F20230413_080253_03723_bk5de-8a431de6-d810-4386-8411-796352eda108.parquet	parquet	April 13, 2023, 09:03:08 (UTC+01:00)	9.0 MB

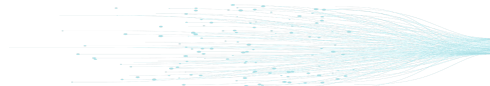
There will also be a 'metadata' folder, containing the various metadata and stats files.

Going back to the output of the SQL viewing the partitions information for the "iceberg"."student1"."sales1" Table.

There is a huge skew of data in the 'country' = 'United Kingdom' partition.

This can be confirmed by running the SQL:

```
SELECT
  COUNT(*) COUNT,
  country
FROM
  iceberg.student1.sales1
GROUP BY
  country
ORDER BY
  1 DESC;
```



count	country
981330	United Kingdom
17866	EIRE
17624	Germany
14330	France
5140	Netherlands
3811	Spain
3189	Switzerland

The UK records account for ~92% of all records. This makes 'country' potentially a bad candidate to partition.

Let us explore an alternative partitioning column – or collection of columns (or hashes of columns) to partition the Table.

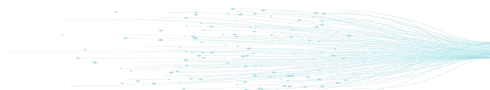
Date and Time are often used as partition columns, the following will display the distribution of records for each month in the data:

```
SELECT
  COUNT(*) COUNT,
  (date_trunc('month', invoicedate)) MONTH
FROM
  "iceberg"."student1"."sales1"
GROUP BY
  (date_trunc('month', invoicedate))
ORDER BY
  1 DESC;
```

The Distribution based on months is reasonable – not perfect, but a candidate. There are 27 x partitions in this dataset (number may change based on student activity).

We could also evaluate each day – this dataset has 606 discrete values, so there would be 606 x partitions. The SQL is below:

```
SELECT
  COUNT(*) COUNT,
  (date_trunc('month', invoicedate)) MONTH
FROM
```



```
"iceberg"."student1"."sales1"  
GROUP BY  
  invoicedate  
ORDER BY  
  1 DESC;
```

Let us create another table to experiment with Partitions. We will create this table as a copy, but this time we will partition by month. The SQL is as follows:

```
CREATE TABLE  
  "iceberg"."student1"."sales3"  
WITH  
  (  
    FORMAT = 'PARQUET',  
    format_version = 2,  
    partitioning = ARRAY['month(invoicedate)'],  
    type = 'ICEBERG'  
  ) AS  
SELECT  
  *  
FROM  
  "iceberg"."student1"."sales1";
```

View the partitions:

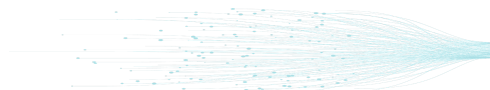
```
SELECT * FROM "iceberg"."student1"."sales3$partitions" LIMIT 50;
```

There are 26 x Partitions in this Table (number can change based on activity).

If the Data Volumes and workloads dictated that more partitions are required – then we can change the partitioning without having to re-create the Table.

For example, we could add 2 x buckets to each partition based on a hash of the values in the 'country' column.

```
ALTER TABLE "iceberg"."student1"."sales3"  
SET PROPERTIES partitioning = ARRAY['month(invoicedate)', 'bucket(country, 2)']
```



If we knew there were queries that were based on a particular month – and filtered on Country, then data will be located in one partition and in a sub-partition bucket file(s).

Again, view the partitions based on the updated partition:

```
SELECT * FROM "iceberg"."student1"."sales3$partitions" LIMIT 50;
```

Add more data to the Table to view the data getting allocated to a particular partition and bucket.

```
INSERT INTO
```

```
"iceberg"."student1"."sales3"
```

```
VALUES
```

```
('123456', '98765', 'Iceberg Badge', 1, DATE('2023-05-11'), 10.2, 222, 'Sweden'),  
( '555555', '44444', 'Starburst Swag', 1, DATE('2023-05-12'), 20.20, 333, 'Sweden'),  
( '777777', '33333', 'Data Jedi T-Shirt', 1, DATE('2023-05-13'), 20.20, 333, 'Finland')
```

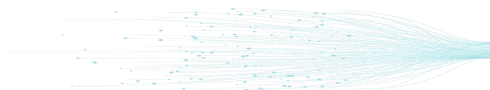
General Comments:

When partitioning by date, a common issue can be the most recent data can be 'hotter' than historic data. This can cause a bottleneck on that one 'hot' partition.

This may not be an issue – it depends on the data access patterns – but it is a consideration when determining Table design.

Partitioning strategy and best practices merits more than can be covered here. There are many considerations – including the workloads that will run against the table, the SQL constructs - such as the joins, filters, aggregates, predicates, and columns selected.

The Volume of the data is another factor here. There should be a balance between the number of partitions, the data distribution, the size of the files, the number of files, avoiding sparse data partitions/files, Query performance, maintenance, etc.



11. Altering Table Metadata

We can change the names of Tables Columns:

```
ALTER TABLE "iceberg"."student1"."sales3" RENAME COLUMN stockcode to sku;
```

Run some SQL to validate the update:

```
SELECT * FROM "iceberg"."student1"."sales3" LIMIT 10;
```

We can add a column to the Table:

```
ALTER TABLE "iceberg"."student1"."sales3" ADD COLUMN category VARCHAR(50);
```

Run some SQL to validate the update:

```
DESCRIBE "iceberg"."student1"."sales3";
```

```
INSERT INTO
```

```
  "iceberg"."student1"."sales3"
```

```
VALUES
```

```
  ('555555', '44444', 'Starburst Swag', 1, DATE('2023-05-12'), 20.20, 333, 'Sweden',  
  'Merchandise')
```

```
SELECT * FROM "iceberg"."student1"."sales3" WHERE category IS NOT NULL;
```

We can drop a column:

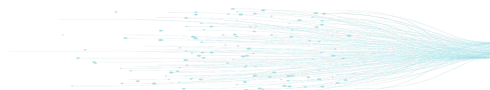
```
ALTER TABLE "iceberg"."student1"."sales3" DROP COLUMN category ;
```

Run some SQL to validate the update:

```
SELECT * FROM "iceberg"."student1"."sales3" LIMIT 10;
```

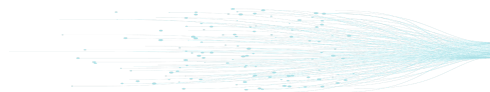
We can rename the Table:

```
ALTER TABLE "iceberg"."student1"."sales3" RENAME TO sales_consume;
```



Run some SQL to validate the update:

```
SELECT * FROM "iceberg"."student1"."sales_consume" LIMIT 10;
```



Appendix A

Links to Starburst Online Documentation

Starburst Homepage:

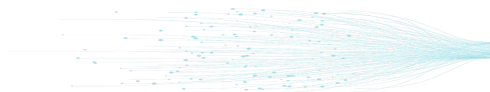
<https://docs.starburst.io/>

Starburst Reference Documentation:

<https://docs.starburst.io/latest/index.html>

Details on S3 access:

<https://docs.starburst.io/latest/connector/hive-s3.html>



Appendix B

Ingesting raw csv data

The bonus exercise below walks through the steps if you were to be given csv files, and how to create an Iceberg based on those files.

Repeat the same steps as you did for creating the Iceberg Catalog – except create a Hive Catalog.

The different steps are:

- Enter a different catalog name, e.g. s3hive
- Default directory name – hive
- Allow External Tables and Writes
- Default Table Format – hive

Create a Schema and then a Hive Table pointing to the csv files in the s3 bucket:

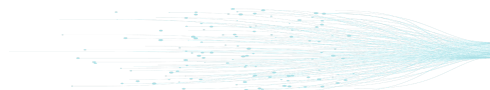
```
CREATE SCHEMA s3hive.student1;
```

```
CREATE TABLE
```

```
s3hive.student1.sales_land (  
  invoice VARCHAR,  
  stockcode VARCHAR,  
  description VARCHAR,  
  quantity VARCHAR,  
  invoicedate VARCHAR,  
  price VARCHAR,  
  customerid VARCHAR,  
  country VARCHAR  
)
```

```
WITH
```

```
(  
  FORMAT = 'CSV',  
  EXTERNAL_LOCATION = 's3://galaxy-data-inovation/retail/',  
  format = 'TEXTFILE',  
  textfile_field_separator = ',',
```



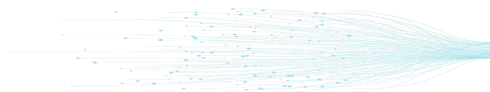

```
    skip_header_line_count = 1  
);
```

CREATE TABLE

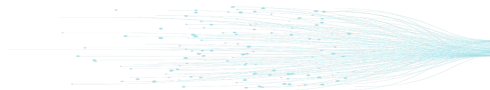
```
iceberg.student1.sales_structure  
(  
    invoice VARCHAR,  
    stockcode VARCHAR,  
    description VARCHAR,  
    quantity INT,  
    invoicedate date,  
    price decimal (8, 2),  
    customerid INT,  
    country varchar  
)  
WITH  
(  
    FORMAT = 'PARQUET',  
    format_version = 2,  
    type = 'ICEBERG'  
);
```

INSERT INTO iceberg.student1.sales_structure SELECT

```
    invoice,  
    stockcode,  
    description,  
    TRY_CAST(quantity as INT),  
    DATE(  
        SUBSTR(invoicedate, 7, 4) || '-' || SUBSTR(invoicedate, 4, 2) || '-' ||  
SUBSTR(invoicedate, 1, 2)  
    ) AS invoicedate,  
    CAST(price as decimal(8,2)),  
    TRY_CAST(customerid as INT),  
    country  
from  
s3hive.student1.sales_land;  
  
analyze iceberg.student1.sales_structure;
```



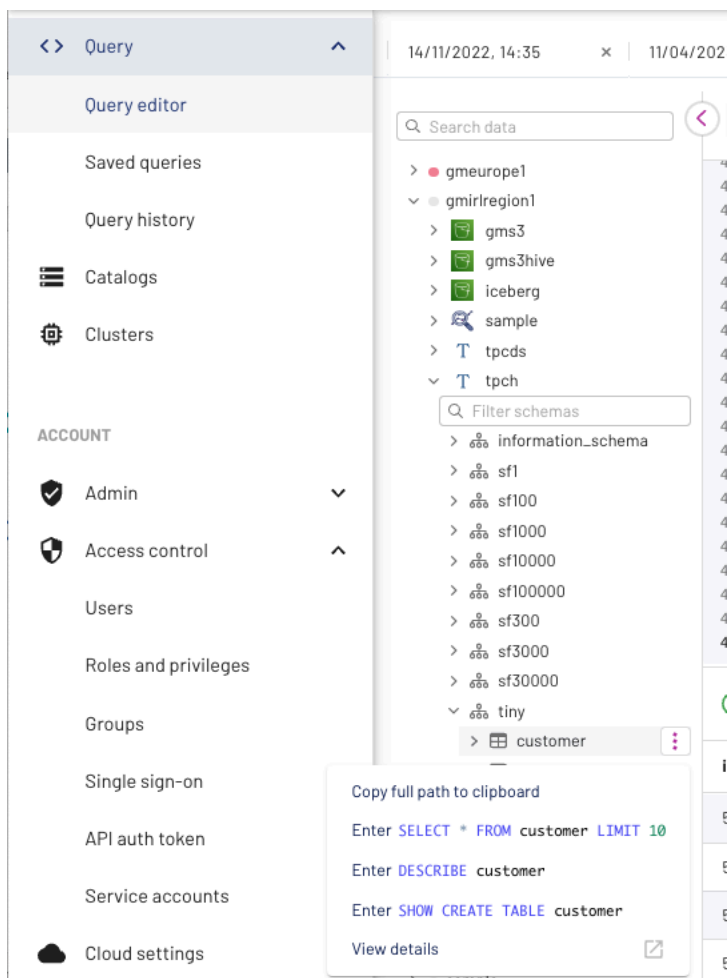
```
show stats for iceberg.student1.sales_structure;  
select * from iceberg.student1.sales_structure limit 10;
```



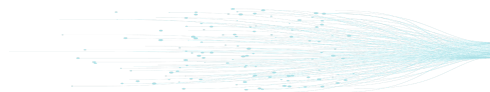
Appendix C

Tips and Tricks

Make use of the three dots drop-down when you highlight a Table:



You can multiple SQL Statements – as long as you have a semi-colon (;) at the end of the statement. The statements will run in sequence.



```
SELECT current_date;
SELECT current_timestamp; -- same as now()
SELECT now();           -- same as current_timestamp
SELECT current_time;
SELECT localtimestamp;
SELECT localtime;
SELECT current_time + interval '1' minute ;
SELECT current_time - interval '2' hour;
SELECT current_date + interval '3' day ;
SELECT current_date - interval '4' month ;
SELECT current_date + interval '5' year ;
SELECT current_timestamp + interval '30' second ;
VALUES now();
```

SQL Examples:

```
SHOW ROLE GRANTS;
```

```
SHOW CATALOGS;
```

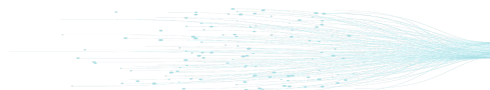
```
SHOW SESSION;
```

```
SHOW FUNCTIONS;
```

```
SHOW STATS FOR "iceberg"."student1"."sales1"
```

```
ANALYZE "iceberg"."student1"."sales1"
```

```
EXPLAIN ANALYZE SELECT * FROM "iceberg"."student1"."sales1"
```



Appendix D

SQL in full

Note: The SQL is written for 'student1'. We recommend to make a Global substitution for the string 'student1' with the value that you have been assigned, e.g. 'student15'.

---- Student Notes from here

---- TPC-H Queries

SELECT

```
COUNT(*) AS LINEITEMS,  
Q.PARTKEY,  
R.NAME,  
P.ORDERKEY
```

FROM

```
tpch.tiny.customer AS R  
INNER JOIN tpch.tiny.orders AS P ON R.CUSTKEY = P.CUSTKEY  
INNER JOIN tpch.tiny.lineitem AS Q ON P.ORDERKEY = Q.ORDERKEY
```

GROUP BY

```
Q.PARTKEY,  
R.NAME,  
P.ORDERKEY
```

HAVING

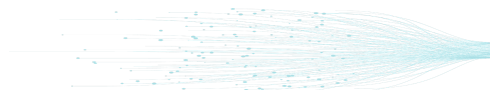
```
COUNT(*) > 1
```

ORDER BY

```
1 DESC;
```

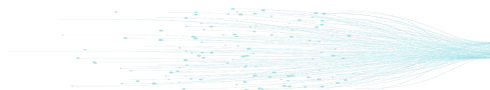
SELECT

```
COUNT(*) AS LINEITEMS,  
Q.PARTKEY,  
R.NAME,  
P.ORDERKEY,  
T.SUPPKEY,  
U.PARTKEY,  
V.NAME,  
W.NAME as NATION,  
X.NAME as REGION
```



```
FROM
  tpch.tiny.customer AS R
  INNER JOIN tpch.tiny.orders AS P ON R.CUSTKEY = P.CUSTKEY
  INNER JOIN tpch.tiny.lineitem AS Q ON P.ORDERKEY = Q.ORDERKEY
  INNER JOIN tpch.tiny.supplier AS T ON Q.SUPPKEY = T.SUPPKEY
  INNER JOIN tpch.tiny.partsupp AS U ON T.SUPPKEY = U.SUPPKEY
  INNER JOIN tpch.tiny.part AS V ON U.PARTKEY = V.PARTKEY
  INNER JOIN tpch.tiny.nation AS W ON T.NATIONKEY = W.NATIONKEY
  INNER JOIN tpch.tiny.region AS X ON W.REGIONKEY = X.REGIONKEY
WHERE
  X.NAME = 'EUROPE'
GROUP BY
  Q.PARTKEY,
  R.NAME,
  P.ORDERKEY,
  T.SUPPKEY,
  U.PARTKEY,
  V.NAME,
  W.NAME,
  X.NAME
HAVING
  COUNT(*) > 1
ORDER BY
  1 DESC;

SELECT
  COUNT(*) AS LINEITEMS,
  Q.PARTKEY,
  R.NAME,
  P.ORDERKEY
FROM
  tpch.sf1.customer AS R
  INNER JOIN tpch.sf1.orders AS P ON R.CUSTKEY = P.CUSTKEY
  INNER JOIN tpch.tiny.lineitem AS Q ON P.ORDERKEY = Q.ORDERKEY
GROUP BY
  Q.PARTKEY,
  R.NAME,
  P.ORDERKEY
HAVING
  COUNT(*) > 1
```



```
ORDER BY
  1 DESC;

--DROP SCHEMA iceberg.student1;

CREATE SCHEMA iceberg.student1
WITH
  (LOCATION = 's3://galaxy-data-inovation/iceberg//student1');

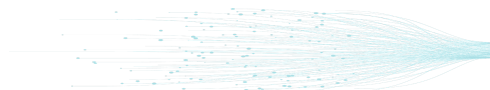
CALL iceberg.system.register_table (
  schema_name => 'student1',
  table_name => 'sales_land',
  table_location => 's3://galaxy-data-inovation/iceberg/student1/'
);

ANALYZE iceberg.student1.sales_land;
SHOW STATS FOR iceberg.student1.sales_land;
SELECT COUNT(*) from iceberg.student1.sales_land;

SELECT
  COUNT(*) count,
  COUNTRY
from
  iceberg.student1.sales_land
GROUP BY
  COUNTRY
ORDER BY
  1 DESC;

--DROP TABLE iceberg.student1.sales1;

CREATE TABLE
  iceberg.student1.sales1
WITH
  (
    FORMAT = 'PARQUET',
    format_version = 2,
```



```

    partitioning = ARRAY['country'],
    type = 'ICEBERG'
) AS
SELECT
*
FROM
    iceberg.student1.sales_land

ANALYZE iceberg.student1.sales1;
SHOW STATS FOR iceberg.student1.sales1;
SELECT COUNT(*) from iceberg.student1.sales1;

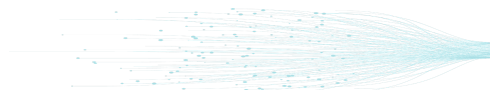
SELECT * FROM "iceberg"."student1"."sales1" LIMIT 10;
SELECT * FROM "iceberg"."student1"."sales1$properties" LIMIT 10;
SELECT * FROM "iceberg"."student1"."sales1$history" LIMIT 10;
SELECT * FROM "iceberg"."student1"."sales1$snapshots" LIMIT 10;
SELECT * FROM "iceberg"."student1"."sales1$manifests" LIMIT 10;
SELECT * FROM "iceberg"."student1"."sales1$partitions" LIMIT 10;
SELECT * FROM "iceberg"."student1"."sales1$files" LIMIT 10;
SELECT *, "$path", "$file_modified_time" FROM "iceberg"."student1"."sales1" ;

INSERT INTO
    "iceberg"."student1"."sales1"
VALUES
    ('999999', '88888', 'Galaxy T-Shirt', 1, DATE('2023-05-11'), 10.2, 777, 'Sweden'),
    ('555555', '44444', 'Cmd BunBun', 1, DATE('2023-05-12'), 20.20, 333, 'Sweden')

SELECT
*
FROM
    "iceberg"."student1"."sales1"
WHERE
    invoice = '494234';
-- invoice = '537434';

UPDATE "iceberg"."student1"."sales1"
SET
    price = price * 1.1
WHERE
    invoice = '494234';

```




```
DELETE FROM "iceberg"."student1"."sales1"
WHERE
  invoice = '494234';
  --invoice = '537434';
```

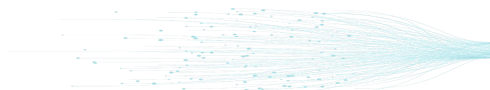
```
CREATE TABLE
  iceberg.student1.sales2
(
  invoice VARCHAR,
  stockcode VARCHAR,
  description VARCHAR,
  quantity INT,
  invoicedate date,
  price decimal (8, 2),
  customerid INT,
  country varchar
)
```

```
INSERT INTO
  "iceberg"."student1"."sales2"
VALUES
  ('123456', '98765', 'Iceberg Badge', 1, DATE('2023-05-11'), 10.2, 222, 'Sweden'),
  ('555555', '44444', 'Starburst Swag', 1, DATE('2023-05-12'), 20.20, 333, 'Sweden')
```

```
SELECT * FROM "iceberg"."student1"."sales2"
```

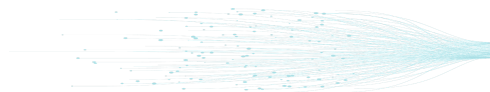
```
SELECT
  a.*,
  b.*
FROM
  "iceberg"."student1"."sales1" a
  INNER JOIN "iceberg"."student1"."sales2" b on a.customerid = b.customerid;
```

```
MERGE INTO "iceberg"."student1"."sales1" AS a USING "iceberg"."student1"."sales2" AS b
ON (a.customerid = b.customerid) WHEN MATCHED
and a.description != b.description THEN
UPDATE
SET
  description = b.description WHEN NOT MATCHED THEN INSERT (
```



```
invoice,  
stockcode,  
description,  
quantity,  
invoicedate,  
price,  
customerid,  
country  
)  
VALUES  
(  
  b.invoice,  
  b.stockcode,  
  b.description,  
  b.quantity,  
  b.invoicedate,  
  b.price,  
  b.customerid,  
  b.country  
);  
  
SELECT *  
FROM  
  "iceberg"."student1"."sales1"  
WHERE stockcode IN ('98765', '44444');
```

```
INSERT INTO  
  "iceberg"."student1"."sales1"  
VALUES  
(  
  '234567',  
  '43210',  
  'trino logo',  
  1,  
  current_date,  
  9.80,  
  678,  
  'Sweden'  
);
```



```
SELECT * FROM "iceberg"."student1"."sales1$snapshots" LIMIT 10;
```

```
SELECT
```

```
  *
```

```
FROM
```

```
  "iceberg"."student1"."sales1"
```

```
--FOR VERSION AS OF 6254229475197879179
```

```
where
```

```
  customerid = 678
```

```
SELECT
```

```
  *
```

```
FROM
```

```
  "iceberg"."student1"."sales1" FOR TIMESTAMP AS OF (current_timestamp - interval '10'  
minute)
```

```
where
```

```
  customerid = 678;
```

```
SELECT
```

```
  COUNT(*) COUNT,  
  country
```

```
FROM
```

```
  iceberg.student1.sales1
```

```
GROUP BY
```

```
  country
```

```
ORDER BY
```

```
  1 DESC;
```

```
SELECT
```

```
  COUNT(*) COUNT,  
  (date_trunc('month', invoicedate)) MONTH
```

```
FROM
```

```
  "iceberg"."student1"."sales1"
```

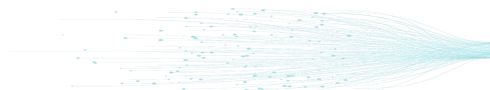
```
GROUP BY
```

```
  (date_trunc('month', invoicedate))
```

```
ORDER BY
```

```
  1 DESC;
```

```
SELECT
```



```
COUNT(*) COUNT,
(date_trunc('month', invoicedate)) MONTH
FROM
  "iceberg"."student1"."sales1"
GROUP BY
  invoicedate
ORDER BY
  1 DESC;

CREATE TABLE
  "iceberg"."student1"."sales3"
WITH
  (
    FORMAT = 'PARQUET',
    format_version = 2,
    partitioning = ARRAY['month(invoicedate)'],
    type = 'ICEBERG'
  ) AS
SELECT
  *
FROM
  "iceberg"."student1"."sales1";

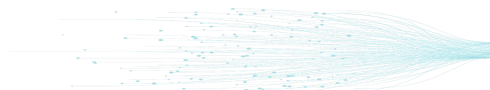
SELECT * FROM "iceberg"."student1"."sales3$partitions" LIMIT 50;

ALTER TABLE "iceberg"."student1"."sales3"
SET PROPERTIES partitioning = ARRAY['month(invoicedate)', 'bucket(country, 2)']

INSERT INTO
  "iceberg"."student1"."sales3"
VALUES
  ('123456', '98765', 'Iceberg Badge', 1, DATE('2023-05-11'), 10.2, 222, 'Sweden'),
  ('555555', '44444', 'Starburst Swag', 1, DATE('2023-05-12'), 20.20, 333, 'Sweden'),
  ('777777', '33333', 'Data Jedi T-Shirt', 1, DATE('2023-05-13'), 20.20, 333, 'Finland')

SELECT * FROM "iceberg"."student1"."sales3$partitions" LIMIT 50;

ALTER TABLE "iceberg"."student1"."sales3" RENAME COLUMN stockcode to sku;
```



```
SELECT * FROM "iceberg"."student1"."sales3" LIMIT 10;

DESCRIBE "iceberg"."student1"."sales3";

ALTER TABLE "iceberg"."student1"."sales3" ADD COLUMN category VARCHAR(50);

INSERT INTO
  "iceberg"."student1"."sales3"
VALUES
  ('555555', '44444', 'Starburst Swag', 1, DATE('2023-05-12'), 20.20, 333, 'Sweden',
  'Merchandise')

SELECT * FROM "iceberg"."student1"."sales3" WHERE category IS NOT NULL;

ALTER TABLE "iceberg"."student1"."sales3" DROP COLUMN category ;

SELECT * FROM "iceberg"."student1"."sales3" LIMIT 10;

ALTER TABLE "iceberg"."student1"."sales3" RENAME TO sales_consume;

SELECT * FROM "iceberg"."student1"."sales_consume" LIMIT 10;

--Sample SQL

SHOW ROLE GRANTS;

SHOW CATALOGS;

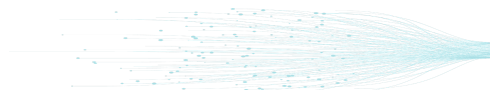
SHOW SESSION;

SHOW FUNCTIONS;

SHOW STATS FOR "iceberg"."student1"."sales1"

--Sample Clean up, don't need to run.

DROP TABLE "iceberg"."student1"."sales1" ;
DROP TABLE "iceberg"."student1"."sales2";
```



```
DROP TABLE "iceberg"."student1"."sales3" ;  
DROP TABLE "iceberg"."student1"."sales_land";  
DROP TABLE "iceberg"."student1"."sales_structure";  
DROP TABLE "iceberg"."student1"."sales_consume";  
DROP SCHEMA iceberg.student1;  
  
DESCRIBE "iceberg2"."student1"."sales_land";
```

