14 Data Sharing

Let's practice creation of the direct data share and all the admin work around (granting needed privileges). There are two options how it can be done. Either there is created a DATABASE ROLE and all the objects which should be part of the share are granted to that DATABASE role, or DB objects are directly granted to the share. Each option has its own use cases where it should be used. You can find more details in documentation.

We are going to try both methods.

Exercise #1 - Data sharing through DATABASE ROLE

1. Create a new database role `SHARE_PROVIDER1` inside `CITIBIKE` DB
2. Grant needed privileges to the new DB role (usage on DB & SCHEMA),
3. We want to share the `TRIPS_MONTHLY` table - grant select to our new DB role
4. Create an empty share `S_MONTHLY_TRIPS`. Do not forget that shares could be created only by ACCOUNTADMIN
5. Grant usage on `CITIBIKE` DB to our newly created share - that's needed prerequisite in order to have access to objects inside
6. Grant the DB role to the share

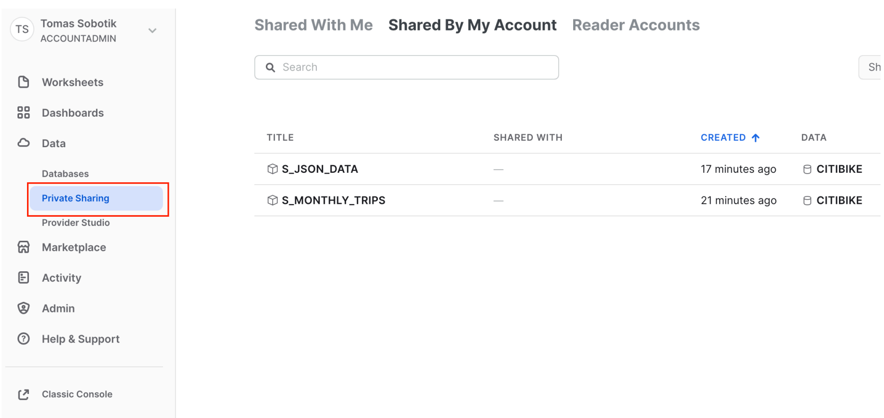
Now everything what is granted to DB role `SHARE_PROVIDER1` is part of the `S_MONTHLY_TRIPS` data share.

Last step would be adding the consumers account to actually share the data with someone. That would be done by `ALTER SHARE` command.

Exercise #2 - Data sharing - granting privileges directly to a share

1. Create empty share S_JSON_DATA
2. Grant needed privileges on the share (usage on DB and schema)
3. This time we want to share our JSON data which means JSON_SAMPLE and JSON_TRIPS_PER_STATION tables. Grant SELECT to share

You can check your shares in UI:



You can also guess when is good to use the first option and when is good to use the second one. If you do not know, documentation will help you.

Bonus - consumer part

If you would like to test also consumer configuration of the share. Then go and create another Snowflake account in the same region as your original account is. Then you can add this new account as a consumer in your in your shares and create a new databases from those shares in the consumer account.

You can find step by step guide in documentation here: https://docs.snowflake.com/en/user-guide/data-share-consumers.html

15 Data Marketplace

Exercise #1 - Starbucks

We are going to get some free dataset from Data Marketplace. Let's try to add a dataset with Starbucks locations and check if we can combine it with our Trips data.

1. Get the SafeGraph - Free POI Data Sample: US Starbucks locations datasets and store id in `STARBUCKS_LOCATION` DB. Apart from `ACCOUNTADMIN` also `SYSADMIN` and `DEVELOPER` roles should have access to that DB.
2. Go and check the `CORE_POI` table from this marketplace dataset. You can try to find out how many Starbucks from NY are there. Send me the count into the chat!

As we have latitude and longitude and we have the same attributes in our TRIPs data set it would be possible to find out how far are the Starbucks cafés from our Citibike stations. For instance we have a Starbuck branch on following street_address: 1095 Lexington Ave

3. Let's find out how many Citibike stations are located in that street. Send me the number into the chat!

BONUS:
If you would like to play with the data more, and you like the geospatial data. You might try to calculate the distance between citibike stations and starbuck cafés.

Exercise #2 - Weather data

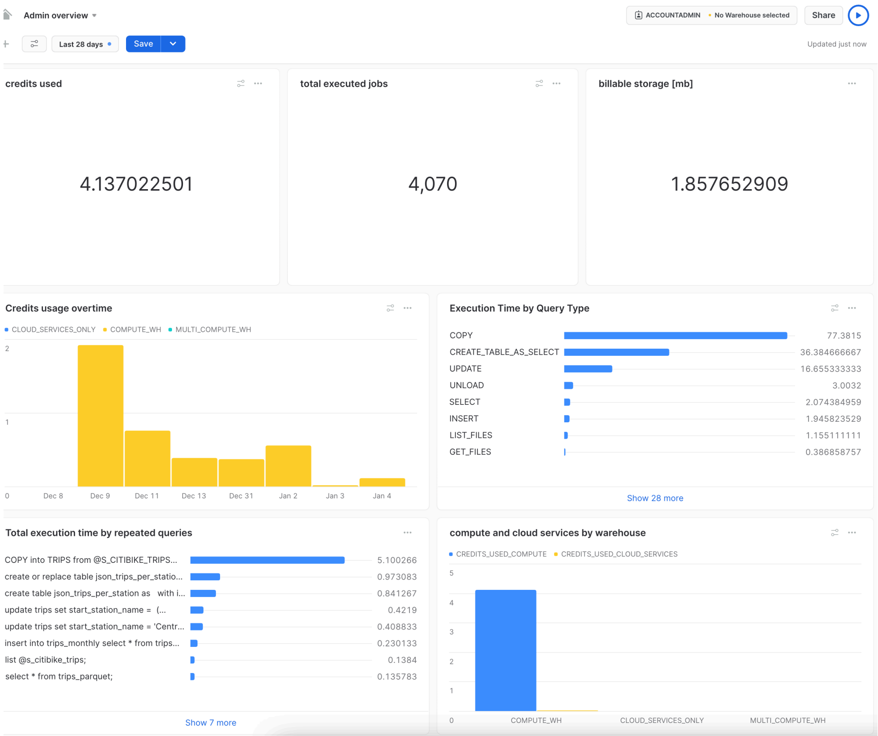
Let's try to add one more dat source from Data Marketplace just to demonstrate how you can easily enrich your data with publicly available datasets. There are plenty of weather related sample datasets. You can try to add some of the Accu Weather datasets to check out how data looks and what attributes it offers.

Which date can you see the forecast for New York? Send me the date to the chat!

16 Snowsight dashboard and Snowflake internal DB

Let's practise two concepts in this exercise. First will be creation of Snowsight dashboards and second one would be related to using of Snowflake internal database which is full of interesting metadata. We are going to create an Admin dashboard which will be visualising data about our account usage - how many credits we have spent, how many jobs have been performed, what are the longest running queries etc.

You can see similar dashboard in following screenshot:



I will provide some queries for the first objects on the dashboard to help you out with syntax and show case what kind of data you can find in `ACCOUNT_USAGE` schema.

1. create a new tile from a worksheet. This one will be showing total used credits in given time frame. As a query use following one:

```
select
    sum(credits_used)
from
    account_usage.metering_history
where
    start_time = :daterange;
```

You can notice special syntax in part related to date time filter. With following syntax `:daterange` you are saying that the real value should be taken from dashboard setting in the runtime.

2. Run the query, toggle in the chart settings and select Scorecard as chart type.
3. Rename the title to **Credits used** and return back to dashboard overview

4. Create a new tile from a worksheet for another dashboard component.
5. This time use following query to get total storage in megabytes

```
select
    avg(storage_bytes + stage_bytes + failsafe_bytes) / power(1024, 3) as
billable_mb
from
```

```
    account_usage.storage_usage
where
    USAGE_DATE = current_date() -1;
```

6. Next steps are same like in the previous example. Toggle in the chart settings, select Scorecard as chart type, rename the worksheet to some more descriptive name and return back to dashboard settings.

7. Last dashboard tile where I will provide a query for you will be for execution time by query type. You can get the data with following query

```
select
    query_type,
    warehouse_size,
    avg(execution_time) / 1000 as average_execution_time
from
    account_usage.query_history
where
    start_time = :daterange
group by
    1,
    2
order by
    3 desc;
```

8. Toggle in chart settings, this time select bar chart, select `AVERAGE_EXECUTION_TIME` column as aggregated one with sum operation, then select `QUERY_TYPE` as Y-Axis, switch to horizontal bars and rename the chart to Execution Time by Query Type.

Now it is time to practise working with `ACCOUNT_USAGE` data on your own. Please try to develop the queries for next dashboard charts by yourself. You can try to create some from the following:

9. Create a scorecard number with total number of executed jobs
10. Create a bar chart showing credit usage over time per virtual warehouse
11. Create a bar chart showing used credits for compute and used credits for cloud services per virtual warehouses
12. Create a bar chart with total execution time per query for repeated queries

If you do not know what kind of view you should use, go and check the documentation where you can find out more details about individual views and what kind of data they offer.

17 Create user defined function

Let's practise creation of user defined function (UDF) which will be using SQL as a language. Our trips dataset contains column called `BIRTH_YEAR`. Write a function called `GET_AGE` which will calculate the current age of the users based on their birth year.

Test the function in the SQL. Write `SELECT` statement using your newly created UDF.

If you are done. You can try to create one more function. We need to check if the bike ride was done during weekend or not. Write a UDF called `WEEKEND_RIDE_CHECK` which will return True in case the bike ride was done during weekend (Saturday or Sunday) otherwise it returns False.

18 Snowpark Python

Exercise 1 - Basics

We are going to practise Snowpark Python in this exercise. You should already have working local development environment for Snowpark and Python. If not, you can find the guide how to prepare it in course prerequisite.

Let's start with connection to our Snowflake account and creation of the session object:

```
from snowflake.snowpark import Session

connection_parameters = {
    "account": "<<your account identifier>>",
    "user": "<<your user>>",
    "password": "<<password>>",
   "role": "SYSADMIN",  # optional
   "warehouse": "COMPUTE_WH",  # optional
    "database": "CITIBIKE",  # optional
    "schema": "PUBLIC",  # optional
}

new_session = Session.builder.configs(connection_parameters).create()
```

Now let's verify the connection by printing the connection details

```
print(new_session.sql("select current_warehouse(), current_database(),
current_schema()").collect())
```

Let's practise little bit a working with data frames. How to create them by different approaches and how to work with them. First, let's create a data frame which will holds our `TRIPS` table.

```
df_trips = new_session.table("trips")
```

And show the first 10 rows

```
df_trips.show()
```

We can also create a data frame from SQL query:

```
df_sql = new_session.sql("select distinct start_station_name from trips
")
df_sql.show()
```

Data transformation. We can filter the data in data frame with `filter()` function:

```
from snowflake.snowpark.functions import col
df_filtered = new_session.table('trips').filter(col("start_station_name")
== 'Front St & Washington St')
df_filtered.show()
```

Or we can select only some columns from table:

```
df_few_columns = df_filtered.select(col('start_station_name'),
col('start_station_id'),col('end_station_name'),
```

```
 col('starttime'), col('tripduration'))

df_few_columns.show()
```

And now the tasks for you.

1. Create a dataset which will contain summary data about trips starting at station called '**Front St & Washington St'.**
Data frame will contain following columns:
- start_station_name
- Month - start time truncated to month level - e.g. 01-01-2023
- trips_count - number of trips in given month
- avg_duration - average trip duration. Value will be rounded to integer value

HINT: You can use session's `sql()` functions for data frame creation and write a SQL command for it.

2. Create a view from the data frame. View will be called '`FRONT_WASHINGTON_SUMMARY`' and it will be stored in `CITIBIKE` DB and `PUBLIC` schema

HINT: You can use data frame's function `create_or_replace_view()` for view creation

3. Query the newly created view via function of your choice and show the the content of it

Exercise 2 - Practising the data frames functions for data transformations

In this exercise we will try to create a same view like in the first exercise but this time we are going to use only data frame's transformation functions (not the SQL command) to select the data.

1. Create a dataset which will contains summary data about trips starting at station called '**Front St & Washington St'.**
Data frame will contain following columns:
- start_station_name
- Month - start time truncated to month level - e.g. 01-01-2023
- trips_count - number of trips in given month
- avg_duration - average trip duration. Value will be rounded to integer value

HINT: You will need functions like `col, count, avg, date_trunc, group_by, agg` …

Please use the Snowpark API reference for checking the syntax or finding the right function. It is available here: https://docs.snowflake.com/ko/developer-guide/snowpark/reference/python/index.html

2. Create a view from the data frame. View will be called 'FRONT_WASHINGTON_SUMMARY' and it will be stored in CITIBIKE DB and PUBLIC schema

3. Query the newly created view via function of your choice and show the the content of it

19 Serverless features

We are going to practise the serverless tasks in this exercise. We already have two tasks as part of our project - T_RIDES_AGG, T_RIDES_LOG. Those tasks uses user managed virtual warehouses. Now we turn them into being serverless.

1. At the beginning, let's clean up the tables which tasks use as targets:

```
truncate table fact_rides;
truncate table log_fact_rides;
```

2. Change the `T_RIDES_AGG` task to be a serverless task. Use `ALTER TASK` command.

3. Check the definition of the task. Now the warehouse attribute should be empty. It means that task is serverless

```
show tasks;
```

4. We try to recreate the other task instead of altering it. In order to be able to create serverless task with SYSADMIN role, we need to grant a new privileges - `EXECUTE MANAGED TASK`:

```
use role accountadmin;
grant EXECUTE MANAGED TASK on account to role SYSADMIN;
use role sysadmin;
```

5. Now we can recreate the task and make it server less by omitting the WAREHOUSE attribute

```
create or replace task t_rides_log
comment = 'Logging the last loaded month'
after T_RIDES_AGG
AS
insert into log_fact_rides select max(month), current_timestamp from
fact_rides;
```

6. Let's test it out now and see what kind of virtual warehouse Snowflake will automatically assign to our task. First we need to resume both tasks

```
alter task t_rides_log resume;
alter task t_rides_agg resume;
```

7. Insert some data into our `TRIPS_MONTHLY` table. Just to repeat. Those data will be part of stream `STR_TRIPS_MONTHLY` and it will automatically trigger our root task and then also the logging task.

```
insert into trips_monthly select * from trips
where date_trunc('month', starttime) = ‘2018-06-01T00:00:00Z';
```

8. Let's check the tables to see if the pipeline has finished.

```
select * from fact_rides;
select * from log_fact_rides;
```

9. If both tables contain the data. We can suspend our tasks now.

```
alter task T_RIDES_LOG suspend;
alter task t_rides_agg suspend;
```

10. Now we can check what virtual warehouse was automatically provisioned by Snowflake for our task run. Let's check the task_history table function to find out the task's query_id:

```
select *
  from table(information_schema.task_history(
  TASK_NAME => 'T_RIDES_AGG'
  ))
  order by scheduled_time desc;
```

11. Find a row with query_id value. It should have also state value = SUCCEEDED. Copy the query_id

12. Let’s look into the `SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY` for query details of given query_id.

```
use role accountadmin;

select * from snowflake.account_usage.query_history where query_id in
('your query id from previous step');
```

You can find in the result set the details about used warehouse. You can see that in my case Snowflake has used the medium size warehouse.

	WAREHOUSE_ID	WAREHOUSE_NAME	WAREHOUSE_SIZE	WAREHOUSE_TYPE
1	54763825	COMPUTE_SERVICE_WH_USER_TASKS_POOL_MEDIUM_0	Medium	STANDARD

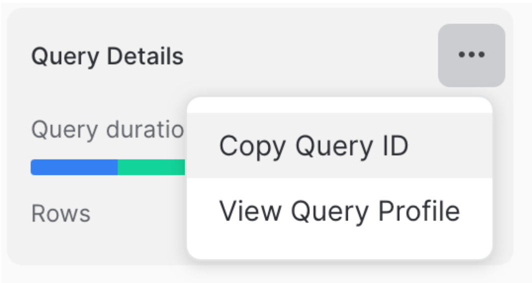
20 Query profile

We are going to work with query profiles in this exercise. To test some recommendations let’s try to run following query where we will apply date filter which should lead to effective prunning.

1. Run following query

```
select * from trips where starttime between to_timestamp('01-01-2017',
'DD-MM-YYYY') and
to_timestamp('30-01-2017', 'DD-MM-YYYY');
```

2. Let’s open the Query Profile through Query Details and have a look on details about the query run. You should be able to see the total amount of partitions, together with info how many of them were really scanned.



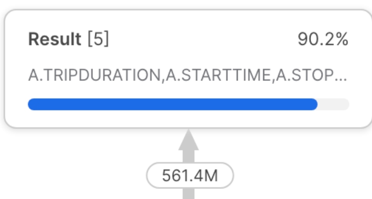
3. As we have only one big table as part of our project. Let’s try to simulate the exploding joins and join the table to itself again. First of all let’s make the warehouse bigger for a while.

```
alter warehouse compute_wh set warehouse_size = xlarge;
```

4. Then run following query. Let it run for approx 2 mins. and then you can cancel it.

```
select * from trips a
join trips b on a.start_station_id = b.start_station_id;
```

5. Go and check the query profile again. You can see how the total amount of records “explodes” after doing this incorrect join



6. Do not forget to scale down the warehouse.

```
alter warehouse compute_wh set warehouse_size = xsmall;
```

7. As a last example we can run following more complex query. Once it is finished you can again open the query profile and see how many operators have been used and how final result has been built.

```
select * from trips where (start_station_name, end_station_name) in (
select start_station_name, end_station_name from trips where
start_station_name in
(
    select start_station_name from trips
        group by start_station_name
        order by count(*) desc
    limit 100

)
group by start_station_name, end_station_name
order by count(*) desc)
;
```

You can see that even though the query aggregates quite a lot of data, it is still possible to do it in memory meaning that there is no data spilling to local or remote disk. That also means that the smallest available warehouse is still sufficient for this operation.