10 Storage related feature

Let's practise the **TIME TRAVEL** and **ZERO COPY CLONING** features. Suppose we accidentally updated some column and now we need to check how data looked before the update and revert the changes back. We can use combination of TIME TRAVEL and ZERO COPY CLONING to fix it.

1. Run the following update statement to simulate the unwanted change:

```
update trips
set start_station_name =
'Central Park S & 6 Ave TMP'
where start_station_id = 2006;
```

2. Use Time Travel and find all rows in table `TRIPS` where value of column `START_STATION_NAME` had the value before we did the update. Use TIME TRAVEL syntax with `AT` keyword.

3. Make a Clone of `TRIPS` table, call it `TRIPS_CLONED`. Content of the table should be from history with initial value for `start_station_name`. Again use TIME TRAVEL SQL syntax in `CLONE` statement

4. Revert the changes in `TRIPS` table. Use either Time travel or your cloned table
5. Drop the cloned table
6. If you want you try `UNDROP` of the cloned table but then drop it again :-)

Send me into the chat how many records you have you updated and TIME TRAVEL syntax you used.

11 Semi structured data

Exercise #1

We are going to practise working with semi structured data. For that purpose we need to first load some JSON data into Snowflake. Let's use some sample JSON data which I have prepared

1. Create a new table for holding the JSON data

```
create table json_sample (value variant);
```

2. Copy the insert statements from file `json_sample.sql` and run them.

3. You should insert 5 records into `JSON_SAMPLE` table. Let's check it

```
select * from json_sample;
```

4. Because the JSON data has quite simple structure without nested elements and arrays, we can easily create a view on top of the JSON file with **colon notation.** Create a view `JSON_SAMPLE_VIEW` which will read data from `JSON_SAMPLE` table.

```
column_name:json_attribute_name::datatype,5. Check the content of the view
```

Exercise #2

We are going to create a JSON structure from relational data in this exercise. It is practice for semi structured data functions like `OBJECT_CONSTRUCT` and `ARRAY_AGG`.
Please use following query as a base dataset for your JSON structure. This query gives you all the trips from station Willoughby St & Fleet St at June 9, 2018.

```
SELECT
    *
    FROM
    TRIPS
    WHERE
    date_trunc('day',starttime) between '2018-06-09' and '2018-06-10'
    AND start_station_id = 239;
```

Please create following JSON structure. The final structure contains nested object and array as well.

```
{
    "StartStationName": << start_station_value >>,
    "day": << start_time_value >>,
    "tripDetails": {
        "duration": << trip_duration_value >>,
        "endStationName": << end_station_value >>
    },
    "userType": [ << all user_type_values for given day and start station >>]
}
```

BONUS:

If the given exercise was too easy for you or you would like to practise it more, you can try to do following, more complex exercise. This time create a JSON structure which will aggregate all the trips in station Willoughby St & Fleet St at June 9, 2018.

The JSON should look like this:

```
{
    "day": << start_time_value >>,
    "stationName": << start_station_value >>,
    "trips": [
    {
        "duration": << trip_duration_value >>,
        "endStation": << end_station_value >>,
        "membershipType": << membership_type_value >>,
        "userType": << user_type_value >>,
        "userbirthYear": << birth_year_value >>
    },
    {
        …. trip #2
    },
    {
       …trip #3
    },
    ]
}
```

Exercise #3

We are going to flatten the semi structured data in this exercise. Before we can begin, we need to prepare suitable, nested, semi structure data set. Please run following script which will create a new table called JSON_TRIPS_PER_STATION. Each row in the table will then contain single JSON document containing all the trips made from given start station at given day.

1. Familiarize yourself with JSON structure
2. Try to flatten the data back into relation form (table) with following columns and define the correct data type for each column:

START_TIME	START_STATION	DURATION	END_STATION	MEMBERSHIP_TYPE	USER_TYPE	USER_BIRTH_YEAR

12 Data governance

Let's create a dynamic data masking policy to mask Personal identifiable information (PII) in trips table.

Values in `BITH_YEAR` and `GENDER` columns should be visible only to admin roles (`SYSADMIN`, `SECURITYADMIN`, `ACOUNTADMIN`) and `ANALYST` role. All others should see masked value. Let's use 0 as masked value.

1. Create a masking policy called `MASK_PII`
2. Mask data in `BIRTH_YEAR` and `GENDER` columns
3. Test it out - try to query the table as `ANALYST` or `SYSADMIN`. You should see unmasked data. Then change the role to `DEVELOPER` and those columns should be masked

Exercise #2

If you managed previous exercise quickly or you want more practice you can try similar thing but this time with usage of tags. Masking policy should be automatically applied to columns which have assigned given tag.

1. Unset the masking policy from `BIRTH_YEAR` and `GENDER` columns
2. Switch to `ACCOUNTADMIN` role
3. Create a tag called `SECURITY_LEVEL`
4. Assign the tag to `BIRTH_YEAR` and `GENDER` columns with tag value = 'PII'
5. Add the `MASK_PII` masking policy to the `SECURITY_LEVEL` tag
6. Test it out - Query the table and used different roles. With `DEVELOPER` role the data should be masked.

13 Streams and Tasks

We are going to practice working with streams and tasks. For that we need to prepare some data. Let's simulate we are receiving the data with trips details on monthly basis. We will create a new table called `TRIPS_MONTHLY`, load there some data and build some aggregation logic on top of the table. It will be using streams to track the changes. Later we will combine it with task to automatically process the records when new data arrive into the table.

1. Create a `TRIPS_MONTHLY` table as a copy of `TRIPS` but it will be empty:
```
create table trips_monthly like trips;
```
2. Create a stream on top of `TRIPS_MONTHLY` table
3. Check the stream details (`SHOW STREAMS` command)
4. Check if stream already has some data (system function called `SYSTEM$STREAM_HAS_DATA()`)
5. Let's insert data into our new `TRIPS_MONTHLY` table. We will use data from April 2018. Run following query to load the data from `TRIPS` table. You should load 1 307 521 rows.
```
insert into trips_monthly
select * from trips
where date_trunc('month', starttime) =
'2018-04-01T00:00:00Z';
```
6. Now the stream should have data. Check it again (`SYSTEM$STREAM_HAS_DATA()`)
7. Try to query the stream like a normal table. You can find the stream metadata columns at the end of the table (last columns).
8. You can check what kind of unique stream action you have in the table.
9. We are going to consume the records from stream. We will be aggregating the records and storing the results in new fact tables for rides. Let's create such table with following script.

```
create table fact_rides (
month timestamp_ntz,
number_of_rides number,
total_duration number,
avg_duration number
);
```

10. Write the aggregation query which would calculate the `NUMBER_OF_RIDES`, `TOTAL_DURATION`, `AVG_DURATION`. As a source you should use the stream `STR_TRIPS_MONTHLY`. Store the result in the new `FACT_RIDES` table.

11. Now the stream should be empty again because we consumed the records in previous step. Let's check it again: `SYSTEM$STREAM_HAS_DATA() function should return FALSE`

12. Send into the chat what is `TOTAL_DURATION` and `AVG_DURATION` for all the trips in April 2018

Exercise #2 - Tasks

Now we are going to turn the logic into the task. We are going the create a task which will take records from the `STR_TRIPS_MONTHLY`, do the aggregation and store the results into `FACT_RIDES` table. Task should be scheduled to run every minute but the code will be triggered only when new data will be placed into the stream.

1. Write a task `T_RIDES_AGG` which with logic and trigger described above.
2. Check the task definition (`SHOW TASKS`)
3. Resume the task. Newly created tasks are suspended and they need to be resumed in order to start operate.
4. Load May 2018 trips data into `TRIPS_MONTHLY` table.
5. Check the `FACT_RIDES`. New row should be inserted there by our task/
6. Send into the chat what is `TOTAL_DURATION` and `AVG_DURATION` for all the trips in May 2018

Exercise #3 - Chaining the Tasks to create a pipeline

Let's create another task which will be running after T_RIDES_AGG. We would have a data pipeline consisting of two steps. Suppose we need some custom logging of the latest loaded month into fact table. For the sake of simplicity we will be just taking the latest loaded month from FACT_RIDES and load it into our custom logging table called LOG_FACT_RIDES. New task should be linked to the T_RIDES_AGG and run after it. T_RIDES_AGG will become a root task of our pipeline.

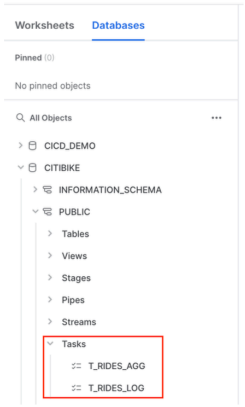
1. Create a log table:

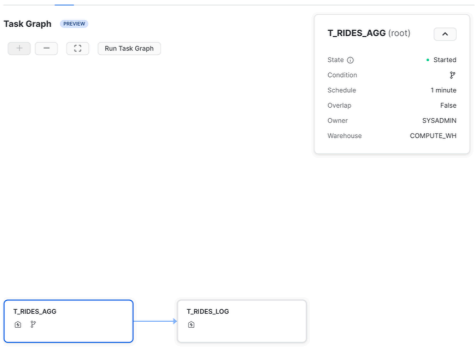
```
create or replace table log_fact_rides
(
    max_loaded_month timestamp_ntz,
    inserted_date timestamp_ntz
);
```

2. Before we can create a new task which will be linked to the `T_RIDES_AGG` we need to suspend it first: suspend the `T_RIDES_AGG` task

3. Create a `T_RIDES_LOG` task:
   - takes max loaded month from `FACT_RIDES` table and store it into `LOG_FACT_RIDES` together with current timestamp
   - run after `T_RIDES_AGG`
4. Resume both tasks
5. Load the June 2018 data into the `TRIPS_MONTHLY` table
6. Check the `FACT_RIDES` and `LOG_FACT_RIDES` tables. Both should be populated with data

from the latest month.
7. Familiarize yourself with new UI related to tasks. You can open the task details page to see the definition, last runs and much more including the DAG visualisation - how the pipeline looks graphically. Go trough those pages to see what everything is available here.





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

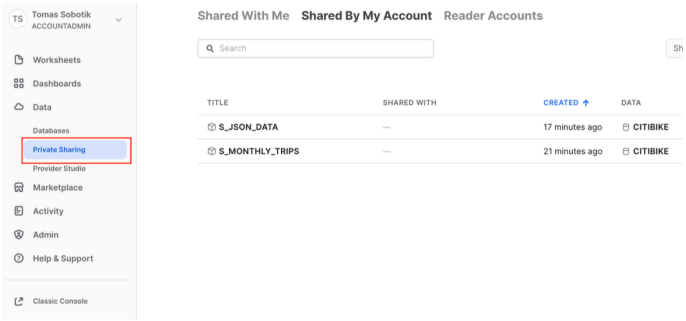
1. Create a new database role `SHARE_PROVIDER1` inside `CITIBIKE` DB
2. Grant needed privileges to the new DB role (usage on DB & SCHEMA)
3. We want to share the `TRIPS_MONTHLY` table - grant select to our new DB role
4. Create an empty share `S_MONTHLY_TRIPS`. Do not forget that shares could be created 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.

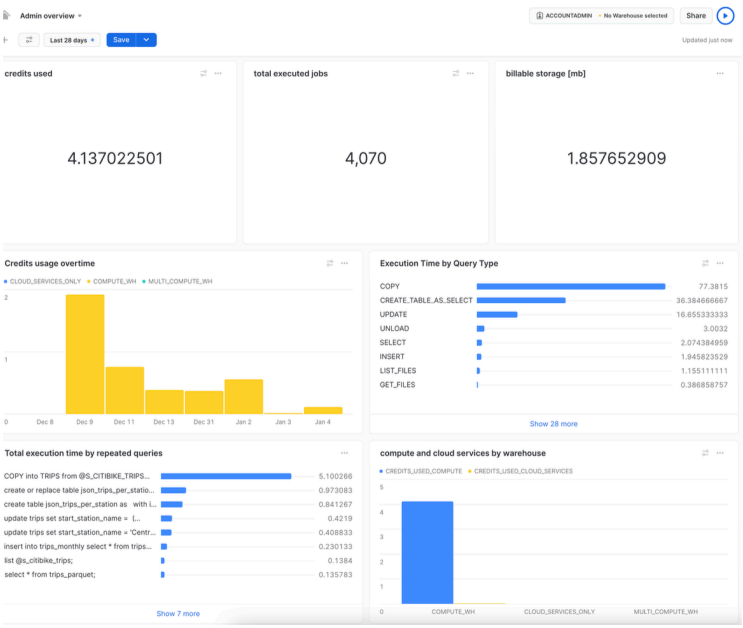
Exercise #2 - Weather data

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

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 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