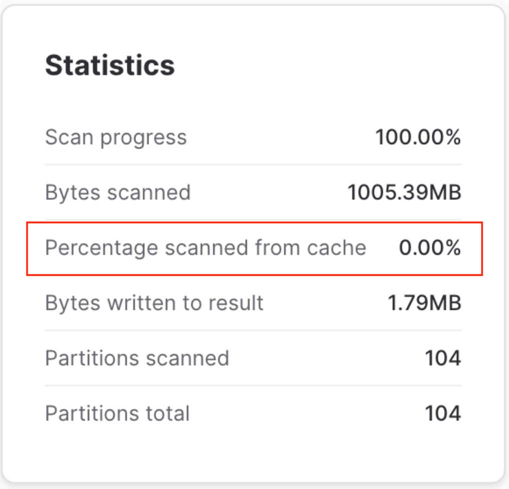
08 Experimenting with Caches



Statistics

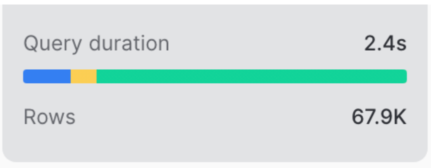
Scan progress	100.00%
Bytes scanned	1005.39MB
Percentage scanned from cache	0.00%
Bytes written to result	1.79MB
Partitions scanned	104
Partitions total	104

In order to have enough time to reuse local cache, let's update the auto suspend time of our warehouse to 5 minutes.

```
     ALTER WAREHOUSE COMPUTE_WH set
AUTO_SUSPEND = 360;
```

1. Let's run following query

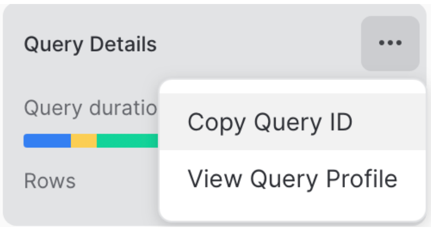
- `SELECT * FROM TRIPS WHERE START_STATION_ID = 3171;`



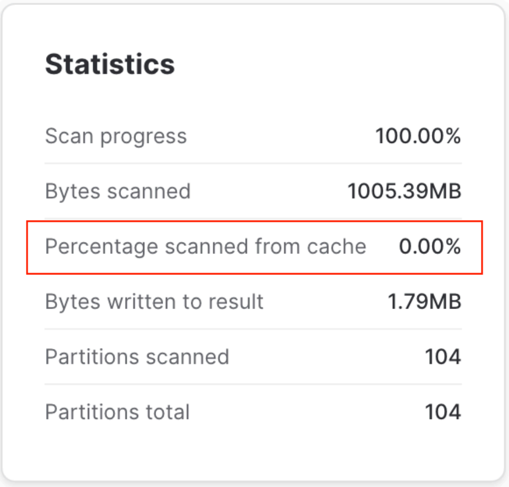
Query duration	2.4s
Rows	67.9K

Look at the right panel to see how long the query ran

At The same place, under the dots (…), you can go into the QUERY_PROFILE, where you can see more details about the query run.



There you can see that nothing has been retrieved from cache.



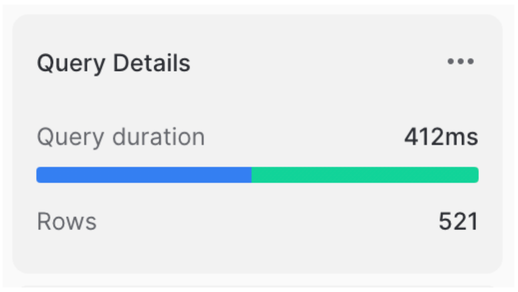
Statistics

Scan progress	100.00%
Bytes scanned	1005.39MB
Percentage scanned from cache	0.00%
Bytes written to result	1.79MB
Partitions scanned	104
Partitions total	104

2. Let's run similar query. Now we want to know what are the most travelled destinations from this station. Let's run following query:

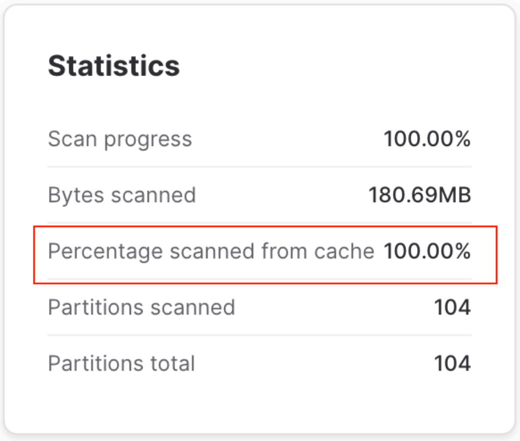
```
SELECT end_station_id, end_station_name, count(*) FROM trips
WHERE start_station_id = 3171
GROUP BY end_station_id, end_station_name
ORDER BY count(*) desc;
```

Have a look on the run time. Even though we have run more complex query which includes also aggregation the run time is more than 5x lower.


Query Details

Query duration	412ms
Rows	521

Let's look again into the query profile, where we can see that result was not fully retrieved from local cache!


Statistics

Scan progress	100.00%
Bytes scanned	180.69MB
Percentage scanned from cache	100.00%
Partitions scanned	104
Partitions total	104

3. Run the first query again:

- `SELECT * FROM TRIPS WHERE START_STATION_ID = 3171;`

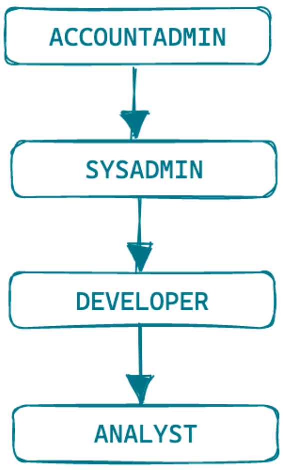
If you open the query profile now, you should see only single node saying that result was retrieved from query result cache.

4. Set the auto-suspend parameter back to 60 seconds now.

```
ALTER WAREHOUSE COMPUTE_WH set AUTO_SUSPEND = 360;
```

5**. BONUS:** How to find out that query result is retrieved from query result cache and thus it is free! Write the response into the chat. :-)

09 RBAC model creation



Let's create couple of custom roles for our project. We will try to follow the best practises, meaning that:

- SECURITYADMIN will be owner of those roles
- SYSADMIN should have access to those roles

Create a role called `ANALYST` with following privileges:

- Read access to table `TRIPS` (SELECT privileges)
- Ability to use warehouse `COMPUTE_WH`

Create a role called `DEVELOPER` with following privileges:

- Access to `ANALYST` Role
- Full access to DB `CITIBIKE`

Use role `ANALYST` and try to create a new table in public schema

Use role `DEVELOPER` and try to create a new table in public schema

Note: If you need to grant access to tables you have to grant USAGE privilege on SCHEMA and DB level.

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
```

```
      },
      ….
      {
       …. trip #N
      },
   ]
}
```

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_YE AR

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()`)
8. Try to query the stream like a normal table. You can find the stream metadata columns at the end of the table (last columns).

9. You can check what kind of unique stream action you have in the table.

10. 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
);
```

11. 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.

12. 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`

13. 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.

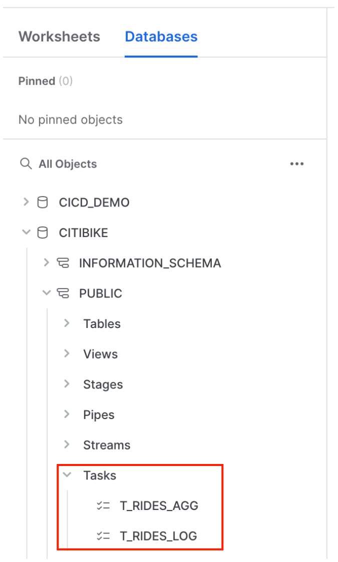
1. Create a log table:

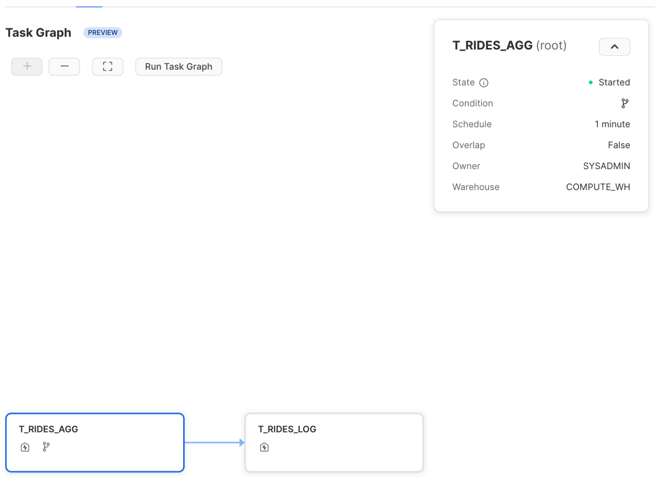
```
create or replace table log_fact_rides
(
max_loaded_month timestamp_ntz,
inserted_date timestamp_ntz
);
```

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.





T_RIDES_AGG (root)

State: Started
Condition
Schedule: 1 minute
Overlap: False
Owner: SYSADMIN
Warehouse: COMPUTE_WH