Understanding Your Snowflake Utilization: Part 3 -- Query Profiling

This lab about query profiling is the third in a three-part series to help you utilize the functionality and data in Snowflake's Information Schema to better understand and effectively Snowflake.

In this lab, I will deep-dive into understanding query profiling. To do this, I will show you examples using the QUERY_HISTORY family of functions. I will also show you a handy page in the UI that provides a graphical view of each query. Keep in mind that you'll need warehouse MONITOR privileges to perform the tasks described in this lab. Typically, the SYSADMIN role has the necessary warehouse MONITOR privileges across your entire account; however, other lower-level roles may also have the necessary privileges.

Ready to get started? Here we go!

Query History Profiling

Query profiling is perhaps one of the more popular topics. Many people are interested in improving their query performance. Although every development team should strive to periodically refactor their code, many find it challenging to determine where to start. Going through this analysis should help with identifying a good starting point.

Let's look at some syntax, per our documentation for [QUERY_HISTORY]:

```
select *
from table(information_schema.query_history(dateadd('hours',-1,
current_timestamp()),current_timestamp()))
order by start_time;
```

This query provides a view into all of the queries run by the current user in the past hour:

row#	QUERY_ID	QUERY_TE...	DATABASE...	SCHEMA_N...	QUERY_TY...	SESSION_ID	USER_NAME	ROLE_NAME	WAREHOU...	WAREHOU...	WAREHOU...	QU
1	29ea94fb-303...	SELECT 1 F...	SALES	PUBLIC	SELECT	49864594794...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
2	eeb9e84a-83...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594711...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
3	ab162ebf-8d1...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594794...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
4	b3c188ff-1cc...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594794...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
5	49077d6c-58...	INSERT INTO...	SALES	PUBLIC	INSERT	49864594794...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
6	9765e6aa-8c...	SELECT 1 F...	SALES	PUBLIC	SELECT	49864594711...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
7	8e62d4c4-8c...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594794...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
8	826c47b9-d7...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594711...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
9	37c75df7-d23...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594752...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
10	e2aaaf91-bdd...	INSERT INTO...	SALES	PUBLIC	INSERT	49864594711...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
11	91faa8b9-611...	SELECT 1 F...	SALES	PUBLIC	SELECT	49864594794...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
12	beabf130-208...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594794...	MARIE	ANALYST_US	NULL	NULL	STANDARD	

We can also leverage the QUERY_HISTORY companion functions to narrow down your focus:

- QUERY_HISTORY_BY_SESSION
- QUERY_HISTORY_BY_USER
- QUERY_HISTORY_BY_WAREHOUSE

These are particularly useful if you have identified specific workflow issues you need to address.

Profiling Tip #1: Using HASH()

Now for a particularly useful tip: utilizing HASH on the QUERY_TEXT column can help you consolidate and group on similar queries (the HASH function will return the same result if any queries are exactly the same). As a general rule, identifying query groups and finding the max and min query runtime should help you sort through specific workflows. In the example below, I'm doing an analysis on average compile and execution time. Additionally, I'm collecting a count of the queries with the same syntax:

```
select hash(query_text), query_text, count(*), avg(compilation_time),
avg(execution_time)
from
table(information_schema.query_history(dateadd('hours',-1,current_timestamp()),current_t

group by hash(query_text), query_text
order by count(*) desc;
```

Output:

row#	HASH(QUERY_TEXT)	QUERY_TEXT	COUNT(*)	AVG(COMPILATION_TIME)	AVG(EXECUTION_TIME)
1	-102792116783286838	SELECT reg_key, looker, created_at, expires_at FR...	36	41.083	3.638
2	-5565463363353937792	SELECT 1 FROM looker_scratch.connection_reg_r...	12	20.000	5.083
3	-8443351143317024671	select hash(query_text), query_text, avg(compilatio...	4	52.000	1940.500
4	3249302775510350827	select * from table(information_schema.query_histo...	2	65.000	1105.000
5	1924843303589736206	select hash(query_text), count(*), avg(compilation_...	1	46.000	566.000
6	-52326745867448573322	select count(*) from json_table;	1	103.000	2.000
7	-1732845353771103631	select * from table(information_schema.warehouse...	1	59.000	299.000
8	-5897400655839230779	select hash(query_text), count(*), avg(compilation_...	1	63.000	720.000
9	2021467874986086700	desc view vw_json;	1	18.000	10.000
10	-79601723214758340004	select json_data:cust_values , json_data:fname::str...	1	31.000	517.000

Using the HASH function further allows a user to easily query a particular instance of this query from the QUERY_HISTORY function. In the example above, I could check for specific queries where the HASH of the query text converted to the value `-102792116783286838`. For example:

```
select *
from table(information_schema.query_history())
where hash(query_text) = -102792116783286838
order by start_time;
```

Output:

row#	QUERY_ID	QUERY_TE...	DATABASE...	SCHEMA_N...	QUERY_TY...	SESSION_ID	USER_NAME	ROLE_NAME	WAREHOU...	WAREHOU...	WAREHOU...	QUI
1	b411f5e2-008...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594795...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
2	984f5f0e-150...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594795...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
3	f2f41772-a46...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594795...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
4	7498c667-68...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594753...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
5	3e83e056-7a...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594753...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
6	b20e5e05-1b...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594795...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
7	59942a92-78...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594795...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
8	1b9d54e0-c9...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594753...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
9	bada7117-bd...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594753...	MARIE	ANALYST_US	NULL	NULL	STANDARD	
10	6c60abe8-ae...	SELECT reg_...	SALES	PUBLIC	SELECT	49864594712...	MARIE	ANALYST_US	NULL	NULL	STANDARD	

The above result shows you all of the times you have issued this particular query (going back 7 days). Pay specific attention to the following columns:

- COMPILATION_TIME

- EXECUTION_TIME
- QUEUED (times)

If a query is spending more time compiling (COMPILATION_TIME) than executing (EXECUTION_TIME), perhaps it is time to review the complexity of the query. Snowflake's query compiler will optimize your query and identify all of the resources required to perform the query in the most efficient manner. If a query is overly complex, the compiler needs to spend more time sorting through the query logic. Take a look at your query and see if there are many nested subqueries or unnecessary joins. Additionally, if there are more columns being selected than required, then perhaps be more specific in your SELECT statement by specifying certain columns.

QUEUED time is interesting because it could be an indicator about your warehouse size and the amount of workload you've placed on the warehouse. Snowflake is able to run concurrent queries and it does a very good job in doing so. However, there will be times when a particularly large query will require more resources and, thus, cause other queries to queue as they wait for compute resources to be freed up. If you see a lot of queries spending a long time in queue, you could either:

- Dedicate a warehouse to these large complex running queries, or
- Utilize Snowflake's multi-clustering warehouse feature to allow more parallel execution of the queries.

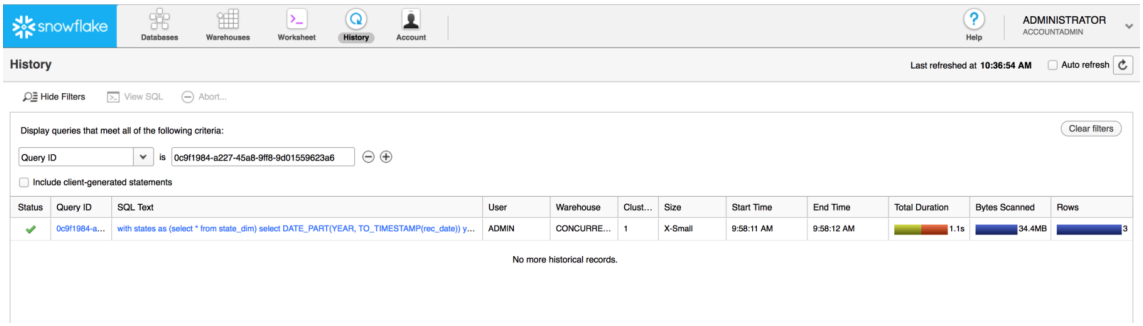
In the recent updates to our QUERY_HISTORY_* Information Schema functions, we have added more metadata references to the results and now you should have a range of metadata at your disposal:

- SESSION_ID
- USER_NAME , ROLE_NAME
- DATABASE_NAME , SCHEMA_NAME
- WAREHOUSE_NAME , WAREHOUSE_SIZE , WAREHOUSE_TYPE

These columns will help you identify the origin of the queries and help you fine tune your workflow. A simple example would be to find the warehouse with the longest-running queries. Or, find the user who typically issues these queries.

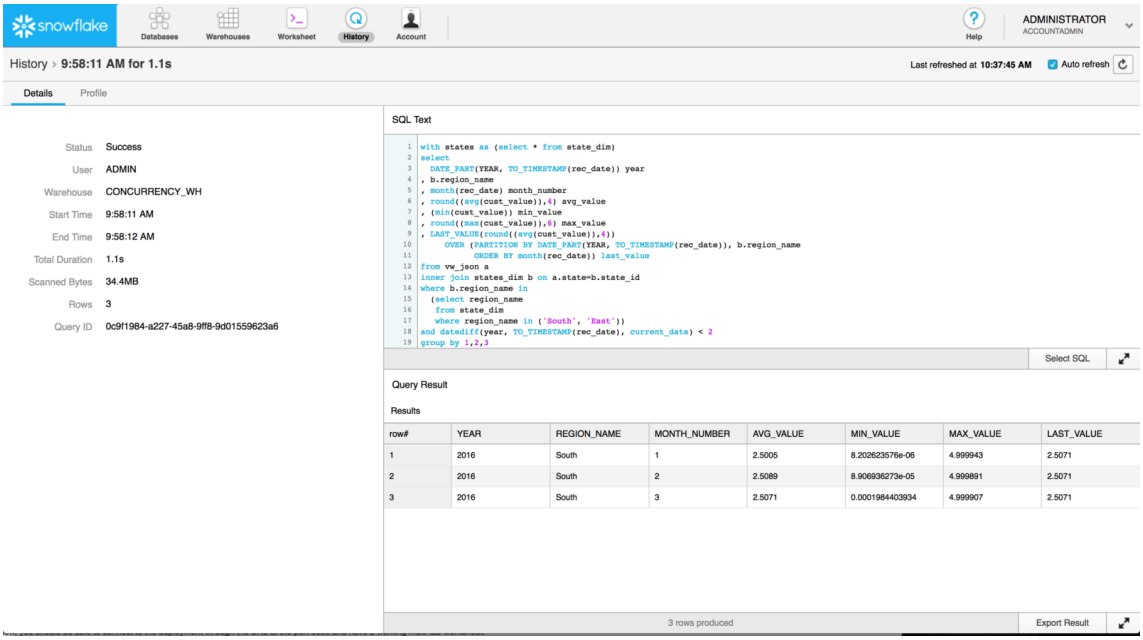
Profiling Tip #2: Using the UI

Once you have identified a particular query you would like to review, you can copy its QUERY_ID value and use this value to view its query plan in Snowflake's Query Profile. To do this, click on the **History** icon, add a QUERY ID filter, and paste the QUERY_ID in question. For example:



Hint: If you don't see a result, make sure you are using a role with the necessary warehouse MONITOR privilege (e.g. SYSADMIN or ACCOUNTADMIN) and you've selected the correct QUERY ID.

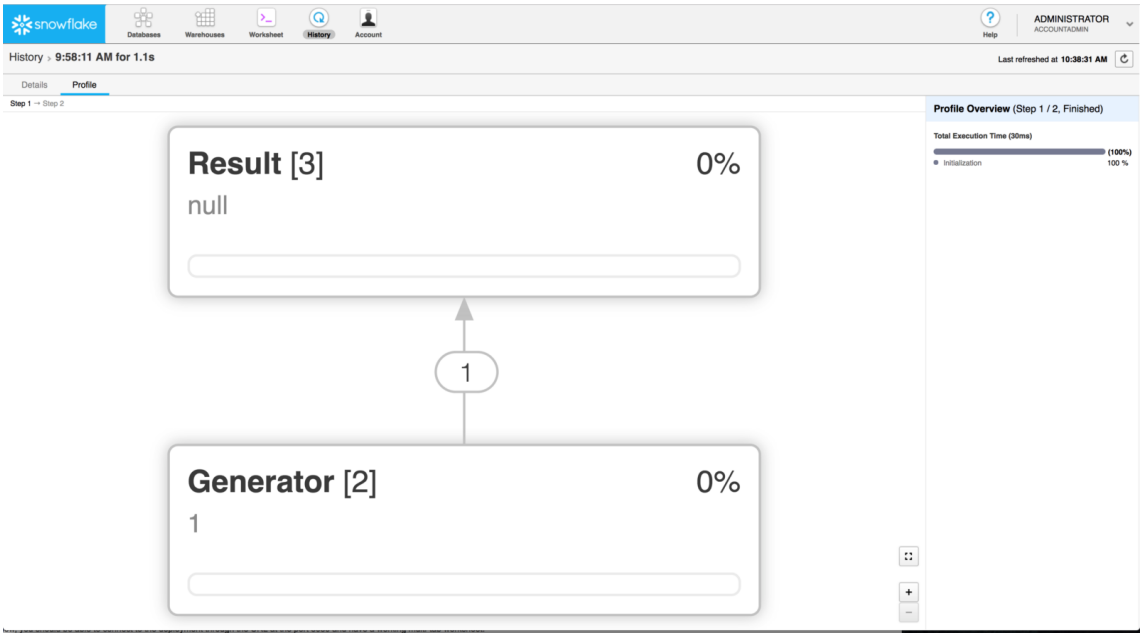
Once the search is complete, you should be able to click on the link provided under the **Query ID** column to go to the query's detail page:



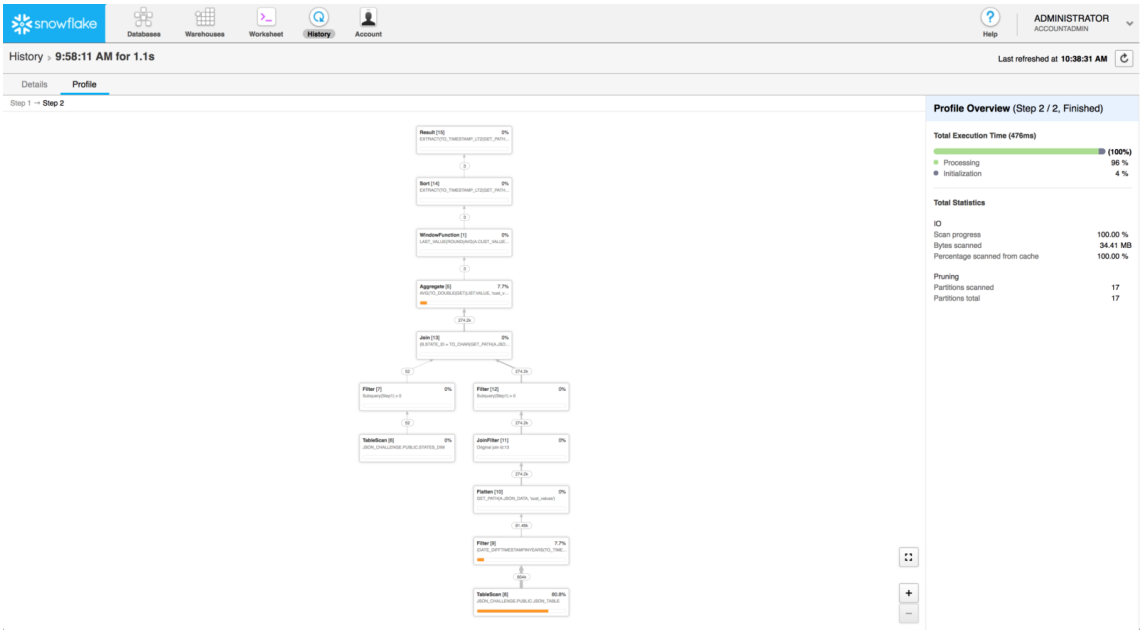
Now click on the **Profile** tab.

You should see a visualization of the Query Profile. In my example, Snowflake shows that this particular query executed in two steps:

Step 1:



Step 2:



Pay particular attention to the ***orange*** bar in this view. It indicates the percentage of overall query time spent on this particular process. In this instance, we can see our query spent most of the time reading data from the table. If we run this query often enough, we should see this time decrease because we'll be reading the data from cache instead of disk.

Conclusion

By utilizing the UI and the Information Schema functions and views described in this lab, you can use query profiling to help you understand your current workflow and identify queries that can be better optimized. This will help save you money in the long run and also improve your user experience. Snowflake will continue to invest in tools like these to help our users better understand and use our platform.