✻ Snowflake logs ingestion and querying cheatsheet ✻

Snowflake SQL basics

Useful links

1. New Web GUI
2. Snowflake SQL reference

Sample data

Logs table structure

DATA	EVENT_TIME	PARSE_TIME
{"user":"testuser","category"..	2022-04-25T10:04:03Z	2022-04-25T10:09:55Z

Data field content

```
{
      "user": "testuser",
      "category": "login",
      "cloud": "AWS",
      "time": "2022-04-25T10:04:03Z",
      "result": "success",
      "ip_address": "10.0.0.1"
}
```

Querying single table

Describe

Describes the structure of the requested item (database, table, etc).

```
describe table DB_NAME.SCHEMA_NAME.TABLE_NAME;
```

Output:

```
name         type              kind
DATA         VARIANT           COLUMN
EVENT_TIME   TIMESTAMP_NTZ(9)  COLUMN
PARSE_TIME   TIMESTAMP_NTZ(9)  COLUMN
```

Fetch all columns from the log table:

```
SELECT *
FROM RAW_LOG_TABLE;
```

Fetch `DATA` column from the log table:

```
SELECT DATA
FROM RAW_LOG_TABLE;
```

Fetch `DATA` sorted by the `EVENT_TIME` column in the default ASCending order

```
SELECT DATA
FROM RAW_LOG_TABLE
ORDER BY EVENT_TIME ASC;
```

Fetch `DATA` sorted by the `EVENT_TIME` column in the default DESCending order

```
SELECT DATA
FROM RAW_LOG_TABLE
ORDER BY EVENT_TIME DESC;
```

Fetch subkey `ip_address` from `DATA` sorted by the `EVENT_TIME` column in the default DESCending order

```
SELECT DATA:ip_address
FROM RAW_LOG_TABLE
ORDER BY EVENT_TIME DESC;
```

Limit/Order

- `ORDER BY` for **large result sets should be used with filtering**, as the command requires to fetch the matching results first to perform the order operation
- `TOP <n>` and `LIMIT <count>` are equivalent (can be used interchangeably)
- `LIMIT <count> [ OFFSET <start> ]`
  can be used interchangeably to:
  `[ OFFSET <start> ] [ ROW | ROWS ] FETCH [ FIRST | NEXT ] <count> [ ROW | ROWS ] [ ONLY ]`

Following queries are equivalent:

```
SELECT DATA:ip_address as ip_address
FROM RAW_LOG_TABLE
ORDER BY EVENT_TIME
LIMIT 100;

SELECT TOP 100 DATA:ip_address as ip_address
FROM RAW_LOG_TABLE
ORDER BY EVENT_TIME;

SELECT DATA:ip_address as ip_address
FROM RAW_LOG_TABLE
ORDER BY EVENT_TIME
FETCH FIRST 100;
```

Aliases

Columns

```
SELECT DATA:ip_address as ip_address
FROM RAW_LOG_TABLE;
```

Tables

```
SELECT DATA:ip_address
FROM RAW_LOG_TABLE as logs;
```

Timestamps

- `TIMESTAMP_TZ` ⇨ timestamp with time zone, converting function: `TO_TIMESTAMP_TZ(string)`
- `TIMESTAMP_LTZ` ⇨ timestamp with local time zone, converting function: `TO_TIMESTAMP_LTZ(string)`
- `TTIMESTAMP_NTZ` ⇨ timestamp with no time zone, converting function: `TO_TIMESTAMP_NTZ(string))`

Time conversion by function

```
SELECT TO_TIMESTAMP_NTZ('2013-04-05 01:02:03');
```

Time conversion by cast operation

```
SELECT '2013-04-05 01:02:03'::TIMESTAMP_NTZ
```

Time operations

Get current timestamp

```
SELECT CURRENT_TIMESTAMP();
```

Add the specified value for the specified date or time

```
SELECT DATEADD(day, -1, current_timestamp());
```

Calculate the difference between two date

```
SELECT DATEDIFF(day, '2021-01-01'::date, '2021-02-28'::date)
```

Filtering the output

Compare

```
SELECT DATA:user
FROM
WHERE EVENT_TIME > DATEADD(day, -1, current_timestamp());
```

Between

```
SELECT DATA:user
FROM
WHERE EVENT_TIME BETWEEN DATEADD(day, -2, current_timestamp())
      AND DATEADD(day, -1, current_timestamp())
```

Not equal

✻ Snowflake logs ingestion and querying cheatsheet ✻

```
SELECT DATA
FROM
WHERE DATA:ip_address != '1.1.1.1';
```

Equal

```
SELECT DATA
FROM
WHERE DATA:result = 'success';
```

Not null

```
SELECT DATA
FROM
WHERE DATA:result IS NOT NULL;
```

In

```
SELECT DATA
FROM
WHERE DATA:result IN ('success', 'failure');
```

Refering to previous queries and results

1. Snowflake stores all query results for 24 hours.
2. Result sets do not have any metadata associated with them, so processing large results might be slower than if you were querying an actual table.
3. The query containing the `RESULT_SCAN` can include clauses, such as filters and `ORDER BY` clauses, that were not in the original query. This allows you to narrow down or modify the result set.
4. A `RESULT_SCAN` is not guaranteed to return rows in the same order as the original query returned the rows. You can include an `ORDER BY` clause with the `RESULT_SCAN` query to specify a specific order.

Retrieve the ID for a last query in the current session

```
SELECT LAST_QUERY_ID();
```

User the result form the last query as a table in new query

```
SELECT *
FROM TABLE(RESULT_SCAN(LAST_QUERY_ID()))
WHERE DATA:result = 'success';
```

Retrieve the values from the `DATA:ip_address` column in the result of the specified query

```
SELECT DATA:ip_address
FROM TABLE(RESULT_SCAN('ce6687a4-331b-4a57-a061-02b2b0f0c17c'));
WHERE DATA:result = 'success';
```

Querying multiple tables

Inner join

`JOIN` (or explicitly `INNER JOIN`) returns rows that have matching values in both tables.

```
SELECT user_signin_logs.username, user_enrichment_logs.position
FROM user_signgin_logs
[INNER] JOIN user_enrichment_logs
ON user_signin_logs.username = user_enrichment_logs.username;
```

user_sigin_logs			user_enrichment_logs	
ip_address	action	username	username	position
10.0.0.1	login	Alice	Alice	Security Engineer
10.100.0.1	login	Bob	Bob	Sales manager
10.100.0.1	login	Victor	James	IT operator

Left join

`LEFT JOIN` returns all rows from the left table with corresponding rows from the right table. If there's no matching row, `NULL`s are returned as values from the second table.

```
SELECT user_signin_logs.username, user_enrichment_logs.position
FROM user_signgin_logs
LEFT JOIN user_enrichment_logs
ON user_signin_logs.username = user_enrichment_logs.username;
```

user_sigin_logs			user_enrichment_logs	
ip_address	action	username	username	position
10.0.0.1	login	Alice	Alice	Security Engineer
10.100.0.1	login	Bob	Bob	Sales manager
10.200.0.1	login	Victor	NULL	NULL

Right join

`RIGHT JOIN` returns all rows from the right table with corresponding rows from the left table. If there's no matching row, `NULL`s are returned as values from the left table.

```
SELECT user_signin_logs.username, user_enrichment_logs.position
FROM user_signgin_logs
RIGHT JOIN user_enrichment_logs
ON user_signin_logs.username = user_enrichment_logs.username;
```

user_sigin_logs			user_enrichment_logs	
ip_address	action	username	username	position
10.0.0.1	login	Alice	Alice	Security Engineer
10.100.0.1	login	Bob	Bob	Sales manager
NULL	NULL	NULL	James	IT operator

Full join

`FULL JOIN` (or explicitly `FULL OUTER JOIN`) returns all rows from both tables if there's no matching row in the second table, `NULL`s are returned.

```
SELECT user_signin_logs.username, user_enrichment_logs.position
FROM user_signgin_logs
FULL [OUTER] user_enrichment_logs
ON user_signin_logs.username = user_enrichment_logs.username;
```

user_sigin_logs			user_enrichment_logs	
ip_address	action	username	username	position
10.0.0.1	login	Alice	Alice	Security Engineer
10.100.0.1	login	Bob	Bob	Sales manager
10.200.0.1	login	Victor	NULL	NULL
NULL	NULL	NULL	Mary	IT engineer

Aggregation and grouping

Group by

`GROUP BY` groups together rows that have the same values in specified columns. It returns summaries (aggregates) for each unique combination of values.

user_sigin_logs		
ip_address	action	username
10.0.0.1	login	Alice
10.100.0.1	login	Alice
10.200.0.1	login	Alice
10.200.0.2	login	Bob
10.300.0.3	login	Bob

count and group by username:

user_sigin_logs	
username	count
Alice	3
Bob	2

Aggregate functions

- `avg(expr)` average value for rows within the group
- `count(expr)` count of values for rows within the group
- `max(expr)` maximum value within the group
- `min(expr)` minimum value within the group
- `sum(expr)` sum of values within the group

Example queries

Find out the number of sign-in log entries:

```
SELECT COUNT(*)
FROM user_signin_logs;
```

Find out the number of sign-in entries with non-null usernames

✻ Snowflake logs ingestion and querying cheatsheet ✻

```
SELECT COUNT(username)
FROM user_signin_logs;
```

Find out the number of distinctive ip_addresses

```
SELECT COUNT(DISTINCT ip_address)
FROM user_signin_logs;
```

Subqueries

A subquery is a query that is nested inside another query, or inside another subquery. There are different types of subqueries.

Single value

The simplest subquery returns exactly one column and exactly one row

This query finds ip_addresses used in sign-in for all the users working as Security Engineers

```
SELECT ip_address
FROM user_signin_logs
WHERE username = (
       SELECT username
       FROM user_enrichment_logs
       WHERE position = 'Security Engineer'
      );
```

Multiple values

A subquery can also return multiple columns or multiple rows. Such subqueries can be used with operators `IN`, `EXISTS`, `ALL`, or `ANY`.

This query finds usernames signed-in for all the IP addresses from data center network.

```
SELECT username
FROM user_signin_logs
WHERE ip_address IN ()
       SELECT ip_address
       FROM user_enrichment_logs
       WHERE network = 'data_center'
      );
```

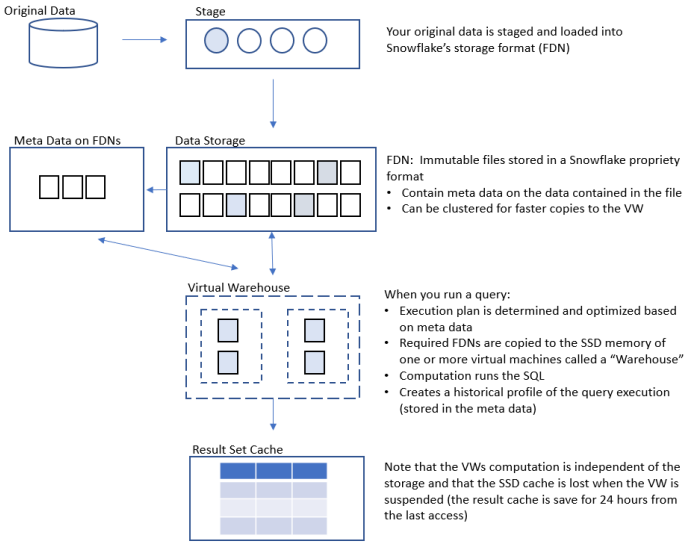
Correlated

A correlated subquery refers to the tables introduced in the outer query. A correlated subquery depends on the outer query. It cannot be run independently from the outer query.

This query finds usernames signed-in if there's a corresponding logs entry in the enrichment table

```
SELECT username
FROM user_signin_logs
WHERE EXISTS (
       SELECT *
       FROM user_enrichment_logs
       WHERE username = user_signin_logs.username
      );
```

Snowflake architecture basics



Data loading

Although very different than storing data on traditional disk, there are many benefits to loading Snowflake data strategically.

1. Sort on ingestion: Data is automatically partitioned in Snowflake on natural ingestion order. Sorting an S3 bucket (using something like syncsort) before bulk load via copy could be way faster than inserting with an `ORDER BY`
2. CSV (gzipped) is the best format for loading to Snowflake (2-3x faster than Parquet or ORC)
3. Use instead of `INSERT` because it utilizes the more efficient bulk loading processes.

Sizing

Take advantage of the native cloud ability to scale, create, and optimize your compute resources.

1. Scale up or out appropriately. As seen above, when you run a query, Snowflake will:
   - Find required FDN files
   - Pull files down into SSD VMs (Note: if 160 GB for AWS or 400 GB for Azure, will **spill over** to remote IO)
   - Performs compute
   - Files will stay on VM **until DW is suspended**
   - Keep in mind: 1 big query = increase size of Data Warehouse
   - VLots of small queries = queries are queuing = increase number of DWs or increase of clusters (if enterprise you can enable multi-cluster)
2. Turn your VW on and off for certain workloads
   - Turn on for batch, then immediately turn off (no reason to wait for *auto-suspend*)
   - Use *auto-resume* when it makes sense
3. Control query processing and concurrency with parameters
   - `max_concurrency_level`
   - Statement queued timeout in seconds
   - Statement timeout in seconds
4. Use warehouse monitoring to size and limit cost per workload (not per database ⇨ this is a shift from the on-prem mentality)
   - If your workload is queuing then add more clusters
   - If your workload is slow with no queuing then size up...

Data modeling

Often overlooked, organizing your information into a mature data model will allow for high-performance SQL scripting better caching potential. Shameless plug, this is Aptitive's bread and butter. Please reach out for a free working session to discuss data modeling for your company.

1. Do a data model for analytics
2. Bake your constraints into design because Snowflake **does not** enforce them
   - Build queries to check for violations
3. Build a process to alert you about loading issues
   - `information_schema.load_history`
   - Aptitive ETL Toolkit

Tracking Usage

Snowflake preserves a massive amount of usage data for analysis. At the very least, it allows you to see which workflows are the most expensive.

✻ Snowflake logs ingestion and querying cheatsheet ✻

1. Use Account Usage Views (eg. `warehouse_metering_history`) for tracking history, performance and cost
2. Don't use `AccountAdmin` or `Public` roles for creating objects or accessing data (only for looking at costs), create securable objects with the *correct* role and integrate new roles into the existing hierarchy
   - Create roles by business functions to track spending by the line of business
3. Use Resource Monitors to cut off DWs when you hit predefined credit amount limits
   - Create one resource monitor per DW
   - Enable notifications

Performance Tuning

The history profiler is the primary tool to observe poorly written queries and make the appropriate changes.

1. Use history profiler to optimize queries
   - The goal is to put the most expensive node in the bottom right hand corner of profiler diagram
   - `system$clustering_depth` shows how effective the partitions are - the smaller the average depth, the better clustered the table is with regards to the specified columns
2. Analyze Bytes Scanned: *Remote* VS *Cache*
   - Make your Bytes Scanned column use *Cache* or *Local* memory most of the time, otherwise consider creating a cluster key to scan more efficiently
3. Make the ratio of partitions scanned to partition used as small as possible by pruning

SQL coding

The number one issue driving costs in a Snowflake deployment is poorly written code! Resist the tendency to just increase the power (and therefore the cost) and focus some time on improving your SQL scripts.

1. Drop temporary and transient tables when done using
2. Don't use `CREATE TABLE AS`, Snowflake hates truncates and reloads for time travel issues. Instead, use `CREATE OR REPLACE`
   - Use `COPY INTO` not `INSERT INTO`
   - Use staging tables to manage transformation of imported data
   - Validate the data **before** loading into Snowflake target tables
3. Use ANSI joins because they are better for the optimizer
   - Use `JOIN ON a.id b.id` format, **NOT** the `WHERE a.id=b.id`
4. Use `WITH` clauses for windowing instead of temp tables or sub-selects
5. Avoid using `ORDER BY`. Sorting is very expensive!
   - Use integers over strings if you must order
6. Don't handle duplicate data using `DISTINCT` or `GROUP BY`

Storing data

Finally, set up the Snowflake deployment to work well in your entire data ecosystem.

1. Locate your S3 buckets in the same geographic region as your Snowflake instance
2. Set up the buckets to match how the files are coming across (eg.by date or application)
3. Keep files between 60-100 MB to take advantage of parallelism
4. Don't use materialized views except in specific use cases (eg.pre-aggregating)

Glossary

1. clone ⇨ a clone is a copy of a storage object (database / schema / table). This is typically a zero-copy clone, meaning the underlying data exists only once but metadata creates 2 different entities on top of the base data.
2. credits ⇨ compute credits are the unit of compute in Snowflake. One credit is charged for one node running for one hour in Snowflake. Larger warehouses consist of more nodes and therefore charge more credits per hour.
3. data sharing ⇨ secure data sharing is a unique feature of Snowflake that allows account-to-account sharing of data. This allows producers to securely expose storage objects (databases / schemas / tables) to consumers. The sharing is live and has a wide range of configurations to ensure the desired billing of storage and compute.
4. database ⇨ is the top-level storage object in Snowflake. All storage objects are contained within a database. This is the highest level of data organization available.
5. file format ⇨ named file format is a collection of rules for processing file data to and from Snowflake stages. File format rules include data formatting, extension-specific options (like skipping headers in CSV files), and error tolerance options (like skipping files with too many errors).
6. materialized view ⇨ a materialized view is a stored query against 1 underlying table (this restriction may change in the future) that automatically runs behind the scenes. The query results are stored (materialized), which can improve read latency.
7. privilege ⇨ privileges are definitions of specific access permissions to specific objects. In Snowflake's security model, privileges on objects are granted to roles. Roles are granted either to users or other roles. Privileges are never directly assigned to users.
8. role ⇨ a role is the unit of Snowflake security to which privileges can be granted to or revoked from. Roles are not users but are assigned to users to authorize user activity.
9. schema ⇨ a schema is the second layer of storage organization in Snowflake below a database. They are containers that hold tables, views, stages, and other bottom-level objects. Security objects and warehouses are not stored at this level. A schema and a database together define a namespace in Snowflake.
10. sequence ⇨ a sequence is a generator object that creates unique values in SQL statements that cover many rows. This is an advanced SQL concept. Check out this article that gives an overview of the concept.
11. Snowpipe ⇨ this refers to Snowflake's continuous loading solution. It is confusing right now because Snowpipe is being upgraded for asynchronous file handling through queues, but not all instances will have this ability (auto ingest). In short, all Snowpipes make regular file ingestion from external stages more manageable for your production workflows.
12. SnowSQL ⇨ SnowSQL refers to the Snowflake CLI tool. It's also commonly used to refer to the actual SQL code that is run in Snowflake.
13. stage ⇨ this is a file location used for data ingestion. Stages can either be internal (managed by Snowflake) or external (managed by you). Stages are just S3 (AWS) or Blob Storage Containers (Azure) where data in Snowflake-supported file formats can be stored before loading into a Snowflake table. Understanding stages is critical to building production data pipelines.
14. stored procedures ⇨ stored procedures are reusable functions defined with a mix of JavaScript and SQL for advanced functionality. These are useful for implementing logic with advanced control flow requirements that are unsupported by SQL (error handling, for-loops, conditional branching).
15. streams ⇨ streams are change records on top of tables. They are queryable like normal tables but include an automatically-updated record of every data change that occurred on the target object. These are a preview feature, so make sure you have it enabled.
16. table ⇨ a table is the lowest level object in Snowflake. It is a structured collection of persisted data.
17. tasks ⇨ a task is a SQL statement executed either on a schedule or in response to the completion of another task. Tasks are useful for job scheduling and are currently preview features.
18. temporary table ⇨ these tables exist only for the duration of a session and are not queriable by any other user. This is useful for ETL processing and helps reduce storage costs as temp tables do not use the same amount of failsafe storage that a standard table does.

✻ Snowflake logs ingestion and querying cheatsheet ✻

19. time travel ⇨ this feature enables users to query data at different points within a range of time (configured at the storage object level). The longer the range of time (up to 90 days, but 1 day by default), the more storage charges are incurred. This feature is valuable for comparing state over time without having to manage additional complex storage structures.
20. transaction ⇨ transaction is a collection of SQL statements that must either be entirely executed successfully or entirely unexecuted (no partial execution). These transactions are fully ACID compliant.
21. transient table ⇨ transient tables are really similar to temporary tables, but they persist beyond a single session and can be queried by other users. They differ from standard tables by having no failsafe storage, making them cheaper but less durable.
22. User defined function (UDF) ⇨ a UDF is a named collection of either SQL or JavaScript logic that accepts arguments and returns either a scalar (single value) or series of rows, depending on how it is defined. It does not support the creation or modification of objects (DML) and only returns newly computed values.
23. user ⇨ a user is an entity of authentication. Authorization is granted to users through roles. Roles are a named collection of 0 or more privileges to perform actions with Snowflake objects. Users are often associated with individuals but are also used to authenticate services, such as BI connections.
24. view ⇨ a view is a table-like object that can be queried but stores no actual data. The structure of a view is defined when it is created as a SQL statement that selects from other underlying objects (including other views).
25. warehouse ⇨ a virtual warehouse is the object of compute in Snowflake. The size of a warehouse indicates how many nodes are in the compute cluster used to run queries. Warehouses are needed to load data from cloud storage and perform computations. They retain source data in a node-level cache as long as they are not suspended. Snowflake credits are billed for a 1-node (XSMALL) warehouse running for 1 hour (10-second minimum charge, prorated per second of run after that).
26. worksheet ⇨ worksheet is a tab within the Snowflake Web UI with its own distinct context from the user's logged-in context. Each worksheet has a SQL editor space where SQL is commonly developed and ran in one location.

The glossary was taken from this page.

Created by Piotr Szwajkowski
piotr.szwajkowski@okta.com