Snowflake Consolidated Notes

Kirk Wilson
Tuesday, March 12, 2019
11:36 AM

NOTE: This is not intended to be a tutorial on Snowflake or provide an overview of its services. Most of this won't make sense if you don't have prior experience with Snowflake and at least a basic comprehension of its architecture. This is solely a collection of, in my opinion, important aspects of Snowflake that are not obvious and are important for anyone on a Snowflake project to be aware of.

All of this information is current as of March 2019.

Data Loading

- Bulk loading of flat files (with COPY command or Snowpipe) is fastest. INSERT with SELECT (or CTAS) is next fastest. Traditional row-by-row inserting is extremely slow (multiple seconds per row).
- Can primarily use two options for loading data:
  - Manually stage the data in internal stage (using PUT command) or external stage. Use COPY INTO to load the target table.
  - Use Snowpipe to support near-real time streaming of data
- COPY command and INSERT statements don't block
  UPDATE, DELETE, and MERGE statements acquire partition locks, hence:
  Use staging tables to manage MERGE
  Consolidate UPDATE and DELETES when possible
- Types of internal Snowflake stages:
  Table Stage: @%[TABLE_NAME]
  Files will be automatically purged if table is dropped
  User Stage: @~
  Personal storage location
  Named Stage: @[STAGE_NAME]
  Probably the best option since you don't have to re-upload a file if you drop a table just for general dev purpose
- A stage can also be created as TEMPORARY, where it exists only within the scope of the Snowflake session in which it was created.
- The PUT command is single-threaded and the only way to parallelize it is with multiple calls in multiple Snowflake sessions. If you have a high volume of files that need to be loaded regularly, loading directly into an external stage with some other mechanism is the only feasible approach.
- When loading from external stage, best practice is to create the stage as a named object and specify the access keys once. Otherwise you have to include the access keys in the COPY INTO statement, which could be a big security risk since those keys are just hanging out in plain text.
- With a PUT command to load an internal stage, Snowflake automatically compresses and encrypts source file(s) on the source machine before any data transfer takes place. Data compression can be controlled, but encryption cannot.

COPY INTO

- Tons of information here: https://docs.snowflake.net/manuals/user-guide/data-load-considerations.html
- When using COPY INTO, best practice is for file size to be between 10 - 100 MB.
- Load multiple files in parallel using pattern matching (Snowflake will automatically parallelize the load process). Alternatively, just manually specify multiple files names in the COPY statement

```
-- Load files from a table's stage using pattern matching to only load uncompressed CSV files
whose names include the string "sales"
copy into mytable
file_format = (format_name = myformat)
pattern='.*sales.*[.]csv';

-- Supports up to 1,000 source files in a single statement
copy into load1 from @%load1/data1/ files=('test1.csv', 'test2.csv', 'test3.csv');
```

- COPY requires an active running warehouse. One file is loaded per thread on the warehouse:
  - X-Small warehouse: 1 node, 4 physical cores, 8 hyperthreads. Thus, a dedicated X-Small warehouse can load 8 files in parallel.
  - Each step up in warehouse size doubles the processing power (Small WH = 16 threads, Medium = 32 threads, etc.)
  - If a larger size warehouse is used to support parallel loading, but the number of files doesn't fit evenly within the number of threads of the warehouse, you're potentially wasting resources (and money). Snowpipe might be a better option due to how it's billed.
- Snowflake does allow for transformations while loading data, including scalar functions and flattening of VARIANT data: https://docs.snowflake.net/manuals/user-guide/data-load-transform.html
- Can't join, filter, or aggregate data during load.
- Use the VALIDATE function to view details of errors that occurred during file load process. Note that this only exists while the staged data still exists - if you automatically purge the source file(s) (PURGE=TRUE) or use the REMOVE command first then VALIDATE will not return anything.

```
SELECT * FROM TABLE(VALIDATE(<table_name>, JOB_ID => '_last'))
```

- Alternatively use VALIDATION_MODE to validate source input without actually loading any data (not supported for COPY statements that transform data during a load)

```
copy into mytable validation_mode = 'RETURN_ERRORS';
```

- Normally Snowflake will not re-load a file that it has already processed (based on file checksums). Add FORCE=TRUE to the COPY command to make Snowflake reload it.
- Uncompressed CSV is the fastest source format
- Can query staged files directly: https://docs.snowflake.net/manuals/user-guide/querying-stage.html
- Use METADATA$FILENAME and METADATA$FILE_ROW_NUMBER to get details on source files. https://docs.snowflake.net/manuals/user-guide/querying-metadata.html
- When loading data into a stage, be sure to consider the stage directory structure if there will be a large number of files.
  - Let's say you have a large number of files to be loaded, segregated by geography. In the COPY INTO statement, Snowflake can more quickly find files in a /US/WA folder than having to do pattern matching on a large number of files in a /US folder that has data for all states.

Snowpipe

- https://docs.snowflake.net/manuals/user-guide/data-load-snowpipe.html
- Optimal source file size is between 1 - 10 MB.
- With pipe loading, you only pay for what you use, rather than having to pre-allocate a specific warehouse size like with COPY and potentially wasting threads if the number of files doesn't exactly match the warehouse size.
- Documentation currently isn't clear if the best practice is to use something like AWS Lambda (https://docs.snowflake.net/manuals/user-guide/data-load-snowpipe-rest-lambda.html) or AWS SQS (https://www.snowflake.com/blog/your-first-steps-with-snowpipe/) to automate data loading with Snowpipe.
    - Based on my testing, the Lambda setup guide seems to have some typos and I wasn't able to immediately get it to work, but I wasn't able to spend any significant time getting deeper into it.
    - The SQS method requires creating the pipe with AUTO_INGEST=TRUE, which is a preview feature and isn't currently enabled in my test Snowflake instance.

Misc. Interesting Bits

- Tons of useful data in INFORMATION_SCHEMA. A few of my favorites are below but the full list is at: https://docs.snowflake.net/manuals/sql-reference/info-schema.html
    - OBJECT_PRIVILEGES to see who (grantor) granted what privilege to who (grantee) and some other interesting bits
    - APPLICABLE_ROLES to show one row for each role grant.
    - TABLE_PRIVILEGES to show what role(s) have access to which tables
    - LOAD_HISTORY to view bulk loading activity
    - TABLE_STORAGE_METRICS - must be ACCOUNTADMIN to query
    - All of the table functions are useful. Just look at the documentation at the link above.
- Can grant FUTURE privileges in a schema. This means any new object created in that schema will automatically have the privilege(s) granted.
    - This is extremely useful if you regularly create new objects, recompile views, etc. as it eliminates the need to constantly update grants.
    - https://docs.snowflake.net/manuals/sql-reference/sql/grant-privilege.html#future-grants-on-schema-objects
- There are many very useful parameters in Snowflake. Below are a few I found interesting but the full list is at https://docs.snowflake.net/manuals/sql-reference/parameters.html
    - STATEMENT_TIMEOUT_IN_SECONDS - lets you specify the max query execution time (can be set at the account and/or warehouse level)
        - Default timeout is TWO DAYS! Use this parameter if you give ad hoc query access to users you don't fully trust.
    - LOCK_TIMEOUT - specifies the number of seconds to wait while trying to lock a resource before timing out
        - Default is 12 hours. Typically only applies to MERGE/UPDATE statements.
    - AUTOCOMMIT - by default statements are auto-committed in Snowflake but it can be disabled if required
    - DATA_RETENTION_TIME_IN_DAYS - even in Enterprise and higher editions, time travel defaults to 1 day unless explicitly increased
    - MAX_CONCURRENCY_LEVEL - gives you control over when new clusters are deployed in multi-cluster warehouse

    - NETWORK_POLICY - only a single policy can be active at a time (for blacklisting/whitelisting IP addresses)
    - QUERY_TAG - string that lets you easily tag a query. See more details below in the Performance Tuning section.
    - QUOTED_IDENTIFIERS_IGNORE_CASE - can disable case sensitivity even for quoted identifiers
    - ROWS_PER_RESULTSET - specifies the max number of rows returned in a result set. See more details below in the Performance Tuning section.
    - SIMULATED_DATA_SHARING_CONSUMER - when setting up data shares, lets you simulate being in the consumer account for testing.
    - STATEMENT_QUEUED_TIMEOUT_IN_SECONDS - how long a query is queued before it's automatically cancelled. This parameter can be used in conjunction with the MAX_CONCURRENCY_LEVEL parameter to ensure a warehouse is never backlogged.
    - TIMEZONE - specifies the time zone for the session (default for all sessions regardless of location is 'America/Los_Angeles').
    - TWO_DIGIT_CENTURY_START - If you encounter date values that only specify two digits for the year ('08' instead of '2008' for example), read the documentation on this.
    - WEEK_OF_YEAR_POLICY and WEEK_START can be useful in certain situations. https://docs.snowflake.net/manuals/sql-reference/functions-date-time.html#label-calendar-weeks-weekdays
- All Snowflake accounts are automatically and transparently updated and within 6 weeks of the current version
- Stored procedures (currently in private beta, but they're not a secret) are written in Javascript
- An important security-related concept to remember is that users never own objects. Only roles own objects. Additionally, user can only assume a single role at a time.
- Although Snowflake supports various constraints, the only enforced constraint is NOT NULL. Other constraints only exist for support with external tools, such as ERD creators.
- Data sharing
    - Reader accounts
        - Used to share data to someone who doesn't have a Snowflake account. Any queries against this goes against your own account.
        - Don't have any license account with Snowflake (or Snowflake support access)
    - Between two Snowflake customers
        - Can create inbound (to receive) and outbound (to share) rules to share data with other Snowflake accounts. Queries are executed against each customer's own warehouse (and costs are incurred accordingly), regardless of who owns the data.
        - Data provider has to create outbound share, and then data receiver has to create inbound share to actually see it.
    - For the more secure versions of Snowflake, data can only be shared to other Snowflake accounts on the same secure edition.
    - Data sharing currently requires Snowflake customers to be on the same provider (AWS or Azure) and in the same region. They are working on removing this limitation eventually though.
- Statistics automatically stored in Snowflake's global services layer allow you to immediately get some data without incurring compute costs (as long as there is not a WHERE clause in the query):
    - COUNT
        - Side note - COUNT(*) and COUNT(1) are identical to the optimizer in Snowflake

    - Number of NULLs for each column
    - MIN/MAX for each column
    - Number of distinct values for each column
- File formats can be stored as:
    - A named object in a database.schema
    - Associated with an internal or external stage
    - Associated with an individual table
    - Associated with a COPY INTO statement
    - Associated with a PIPE
- Table types - https://docs.snowflake.net/manuals/user-guide/tables-temp-transient.html#comparison-of-table-types
    - TEMPORARY - Only exist within the session in which they were created only persist only for the remainder of that session. Not visible to other users or sessions.
        - Note that it is possible to create a temp table with the same name as a transient/permanent table. While the temporary table exists, it effectively hides the non-temp table and any queries/operations on the table will only affect the temp table and not the non-temp table. This can be confusing if you forget about it or accidentally create a temp table with the same name as an existing object.
    - TRANSIENT - Same as a permanent table, except it has no fail-safe period and regardless of the Snowflake edition, it has a max time travel retention period of 1 day. Useful for things like staging table.
    - PERMANENT - Normal table with standard time travel and includes 7 day fail-safe period.
- Use the AS statement to alias a column and reference that alias name in other parts of the query (including WHERE and GROUP BY clauses). This doesn't alter the query execution in any way, but can make writing queries much simpler and improves readability.

```
SELECT
  COL1 || '_SOME_APPENDED_TEXT' AS COLUMN_ALIAS
FROM DUMMY_TABLE
WHERE COLUMN_ALIAS = 'a_SOME_APPENDED_TEXT';
```

- Can script out an entire database using GET_DDL. https://docs.snowflake.net/manuals/sql-reference/functions/get_ddl.html
- Use CHECK_JSON function to validate source data before ingestion.
- Use secure views to give very specific visibility to data that a user/role might otherwise not have access to: https://docs.snowflake.net/manuals/user-guide/views-secure.html. Secure views run as the user who created the view, not as the person who is querying the data. These are generally used for data sharing, but can be useful in other scenarios too.
- Roles best practices:
    - All roles should be created using SECURITYADMIN role
    - All roles should be granted to SYSADMIN
- Don’t forget GRANT USAGE on warehouses/databases/schemas when creating a new role
- Snowflake has CREATE OR REPLACE or CREATE … IF NOT EXISTS so you shouldn't ever have to drop an object first if it already exists just to avoid script errors.
- Don't forget about UNDROP to instantly recreate objects in place if needed
- A row can't be larger than 16 MB (the same max size as a micropartition)
- If timing is ever an issue with queries and you just need a stable data set, use time travel to query as of a specific time (such as the top of an hour).
- Can use functions like ARRAY_AGG, OBJECT_AGG to generate JSON from data in Snowflake

- XML support will (likely) always be in beta. Snowflake has basic support, but not super robust compared to some other tools (large level of effort required on Snowflake's part to implement fully with relatively low customer demand from their perspective)
- Using FLATTEN function on VARIANT data, via a view, can incur extra performance penalty, so you might want to make it a materialized view for large data volumes
- Data unloading is effectively the opposite of loading. COPY INTO from a table into a staging area, and then use GET instead of PUT to download the file. Alternatively, you can unload directly to S3 or Azure Blob.
- Materialized views are always update to date, even if the underlying MV hasn't been refreshed. You query it when there have been some changes to the underlying data since the last refresh, it will query the MV, and then effectively UNION or re-play any changes to data to get the current result set. It won't be necessarily as performant, but data will always be up-to-date and it's still better than a traditional view
- A connection itself (ODBC, Python, …) can have a default role

Misc. How-To

Note that these are just examples to give you ideas of how to do things. It's far from a comprehensive list.

Change ownership of object

```
GRANT OWNERSHIP ON DATABASE <database_name> TO ROLE SECURITYADMIN COPY CURRENT GRANTS;
```

Revoke a role

```
REVOKE ROLE FINANCE_ROLE FROM ROLE PAYROLL_ROLE;
```

Reset a locked account

```
USE ROLE SECURITYADMIN;
 ALTER USER SVC_REPORTING SET MINS_TO_UNLOCK= 0;
```

View details of the Snowflake instance

NOTE: Many of these commands require ACCOUNTADMIN or SECURITYADMIN access to execute
ANOTHER NOTE: You can query the output from the below "SHOW" statements using this query after executing the SHOW statement:

```
SELECT * FROM TABLE(RESULT_SCAN(LAST_QUERY_ID())); -- Note that column names are case-sensitive and must be quoted when used in a filter
SHOW USERS;
SHOW ROLES;
SHOW SCHEMAS;
SHOW DATABASES;
SHOW WAREHOUSES;
SHOW GRANTS OF ROLE <role name>; -- View users/roles assigned to a role
SHOW GRANTS TO USER <user name>; -- View grants assigned to a specific user
SHOW GRANTS TO ROLE <role name>; -- View grants assigned to a specific role
```

Some of this data is also accessible via INFORMATION_SCHEMA, but I feel it's easier to understand what's happening with the above commands.

Get DDL of an object

```
SELECT GET_DDL('FUNCTION', 'ARCHIVE.PUBLIC.DECIMAL_SCALE(VARCHAR)');
SELECT GET_DDL('VIEW', 'ARCHIVE.CERTIFY.EXPENSE_APPROVALS');
```

View warehouse usage

```
SELECT
*
FROM
TABLE(INFORMATION_SCHEMA.WAREHOUSE_METERING_HISTORY(DATEADD('DAYS',-
10,CURRENT_DATE())))
where warehouse_name like 'XXXXXXXX';
```

View delta of before and after loading data

```
SELECT
    (SELECT COUNT(*) FROM <TABLE> AT(OFFSET => -60 * 10)) BEFORE_ETL, -- data as it was 10
    minutes ago
    (SELECT COUNT(*) FROM <TABLE>) AFTER_ETL;
```

View changed data

```
-- View any new or updated records (except for deletes)
SELECT * FROM <TABLE>
MINUS
SELECT * FROM <TABLE> AT (statement => 'query_id'); -- or timestamp
```

Use time travel to view data before a specific query

```
SELECT * FROM TABLE BEFORE (STATEMENT => 'query_id');
```

Performance Testing

- Snowflake is multi-tenant. In order to measure performance, you need to run the test multiple times and take an average. An analogy is "how would you measure the time it takes to ride an elevator from the bottom floor to the top floor?".
- Don't get Snowflake performance measurements corrupted with network performance.
    - Example: running an on-prem Tableau server pulling large result sets across the internet.

- Disable result set caching

  ```
  ALTER SESSION SET USE_CACHED_RESULT = FALSE;
  ```
- Control the size of the returned result set.

  This is different from LIMIT in that the Snowflake optimizer would recognize the LIMIT clause and adjust the execution plan accordingly. This session parameter instead tells Snowflake to execute the query as if it were going to return the entire data set (which is what we want for performance testing), but still only return a subset of the total records to reduce the amount of network overhead.

  ```
  ALTER SESSION SET ROWS_PER_RESULTSET = <number>;
  ```
- Use query tags to make queries easier to find in the query history view in the Snowflake UI.

  ```
  ALTER SESSION SET QUERY_TAG = 'my performance test query';
  ```

```
-- Retrieve isolated Snowflake execution metrics and query_id from the last day for the matching
query tag
select *
from table(information_schema.query_history(dateadd('days',-
1,current_timestamp()),current_timestamp()))
where query_tag = 'my performance test query';
order by start_time;
```

Performance Tuning Options

- Prime the cache. If your workflow allows it, you can run queries in the morning and they will be automatically cached for future reuse throughout the day.
  - https://docs.snowflake.net/manuals/user-guide/querying-persisted-results.html
- Rewrite the query to be more efficient
- Increasing warehouse size frequently improves performance, but not always
- Avoid using "SELECT *"
  - This is bad practice for a number of reasons, but primarily it is very inefficient in a columnar store like Snowflake.
- Avoid using ORDER BY unless necessary as it is extremely expensive in Snowflake.
  - Snowflake doesn't have indexes it can rely on
  - Can easily lead to "spilling", which indicates that the primary resource was exhausted and Snowflake needed to start leveraging the next (secondary) resource
    - Memory on a warehouse node would "spill" to local SSD on the node.
    - A local SSD on a warehouse node would spill over into the cloud storage service (S3 or Azure Blob)
  - If sorting is required, try to avoid sorting on lengthy string values and use integers instead if possible.
- The join order in a query should not affect Snowflake's cost-based optimizer (CBO).
  - In extremely rare cases when a query compilation process times out, the CBO can produce an execution plan that is heavily influenced by the join order defined in the query. This is not something that should affect your day-to-day SQL queries, but keep it in mind as a potential factor in case the CBO just isn't doing what it should.
- In some cases of duplicate data, using a WITH clause (aka common table expression or CTE) to de-dupe the data set before joining to another data set can significantly improve performance (by avoiding a join explosion. A similar option is to materialize data into an intermediate temp table, which can also provide a pruning benefit.
- In filters, improve performance and chances of pruning by simplifying expressions and applying functions on filtering values rather than columns:
  - GOOD: … WHERE my_timestamp >= date::TIMESTAMP
    AND my_timestamp < DATEADD(DAY, 1, date2::TIMESTAMP)
  - BAD: … WHERE TO_DATE(my_timestamp) BETWEEN date1 AND date2
- SQL pattern matching (LIKE) is faster than regular expressions
- Pruning is more effective on dates and numbers than strings (especially long strings, such as UUID)
- Use the ANY_VALUE function to improve performance in some GROUP BY queries: https://docs.snowflake.net/manuals/sql-reference/functions/any_value.html#using-with-group-by-statements
- Define explicit cluster keys (if no cluster key is explicitly defined, the table will be clustered by the natural ingestion order of the data)
  - Generally only useful on tables > 1 TB

- 4 or more cluster keys is typically not useful
- Snowflake will automatically rearrange the micropartitions according to the cluster definition, which will incur additional compute cost. However, they say the performance benefits of this are worth it (the combined cost of reclustering a table and the resulting shortened query times is on average less than the cost of the longer queries against an unclustered table).
- Common cluster key recommendations:
  - Predicates (filter criteria) on tables
  - Join columns
  - Keys for the most important dimensions should be listed first
  - When defining cluster keys, keep the following in mind:
    - Check the number of distinct values of the column using the HLL function.
      - https://docs.snowflake.net/manuals/user-guide/querying-approximate-cardinality.html
      - https://docs.snowflake.net/manuals/sql-reference/functions/hll.html
    - Lower cardinality (fewer distinct values) of leading cluster key columns results in cheaper re-clustering
    - High cardinality keys render any subsequent keys useless
    - Use the system$clustering_information function to get details for how a table is currently partitioned and to evaluate other clustering options. https://docs.snowflake.net/manuals/sql-reference/functions/system_clustering_information.html

```
-- Retrieve clustering information for a table
-- The NULL value in the second parameter allows this to work on tables that are un-clustered as well.
select system$clustering_information('SNOWPIPE_DEMO.PUBLIC.DUMMY_TABLE', '(NULL)' );

-- Retrieve just the total number of micro-partitions in a table
select
parse_json(system$clustering_information('SNOWPIPE_DEMO.PUBLIC.DUMMY_TABLE', '(NULL)' )):"total_partition_count"::INTEGER AS NUM_PARTITIONS;
```