Guides > Databases, Tables, & Views > Materialized Views

Working with Materialized Views

ENTERPRISE EDITION FEATURE

Materialized views require Enterprise Edition. To inquire about upgrading, please contact Snowflake Support.

A materialized view is a pre-computed data set derived from a query specification (the SELECT in the view definition) and stored for later use. Because the data is pre-computed, querying a materialized view is faster than executing a query against the base table of the view. This performance difference can be significant when a query is run frequently or is sufficiently complex. As a result, materialized views can speed up expensive aggregation, projection, and selection operations, especially those that run frequently and that run on large data sets.

Note
Materialized views are designed to improve query performance for workloads composed of common, repeated query patterns. However, materializing intermediate results incurs additional costs. As such, before creating any materialized views, you should consider whether the costs are offset by the savings from re-using these results frequently enough.

Deciding When to Create a Materialized View

Materialized views are particularly useful when:

- Query results contain a small number of rows and/or columns relative to the base table (the table on which the view is defined).
- Query results contain results that require significant processing, including:
  - Analysis of semi-structured data.
  - Aggregates that take a long time to calculate.
- The query is on an external table (i.e. data sets stored in files in an external stage), which might have slower performance compared to querying native database tables.
- The view's base table does not change frequently.

Advantages of Materialized Views

Snowflake’s implementation of materialized views provides a number of unique characteristics:

- Materialized views can improve the performance of queries that use the same subquery results repeatedly.
- Materialized views are automatically and transparently maintained by Snowflake. A background service updates the materialized view after changes are made to the base table. This is more efficient and less error-prone than manually maintaining the equivalent of a materialized view at the application level.
- Data accessed through materialized views is always current, regardless of the amount of DML that has been performed on the base table. If a query is run before the materialized view is up-to-date, Snowflake either updates the materialized view or uses the up-to-date portions of the materialized view and retrieves any required newer data from the base table.

Important
The automatic maintenance of materialized views consumes credits. For more details, see Materialized Views Cost (in this topic).

Deciding When to Create a Materialized View or a Regular View

In general, when deciding whether to create a materialized view or a regular view, use the following criteria:

- Create a materialized view when ***all*** of the following are true:
  - The query results from the view don’t change often. This almost always means that the underlying/base table for the view doesn’t change often, or at least that the subset of base table rows used in the materialized view don’t change often.
  - The results of the view are used often (typically significantly more often than the query results change).
  - The query consumes a lot of resources. Typically, this means that the query consumes a lot of processing time or credits, but it could also mean that the query consumes a lot of storage space for intermediate results.
- Create a regular view when ***any*** of the following are true:

- The results of the view change often.
- The results are not used often (relative to the rate at which the results change).
- The query is not resource intensive so it is not costly to re-run it.

These criteria are just guidelines. A materialized view might provide benefits even if it is not used often — especially if the results change less frequently than the usage of the view.

Also, there are other factors to consider when deciding whether to use a regular view or a materialized view.

For example, the cost of storing the materialized view is a factor; if the results are not used very often (even if they are used more often than they change), then the additional storage costs might not be worth the performance gain.

Comparison with Tables, Regular Views, and Cached Results

Materialized views are similar to tables in some ways and similar to regular (i.e. non-materialized) views in other ways. In addition, materialized views have some similarities with cached results, particularly because both enable storing query results for future re-use.

This section describes some of the similarities and differences between these objects in specific areas, including:

- Query performance.
- Query security.
- Reduced query logic complexity.
- Data clustering (related to query performance).
- Storage and maintenance costs.

Snowflake caches query results for a short period of time after a query has been run. In some situations, if the same query is re-run and if nothing has changed in the table(s) that the query accesses, then Snowflake can simply return the same results without re-running the query. This is the fastest and most efficient form of re-use, but also the least flexible. For more details, see Using Persisted Query Results.

Both materialized views and cached query results provide query performance benefits:

- Materialized views are more flexible than, but typically slower than, cached results.

- Materialized views are faster than tables because of their “cache” (i.e. the query results for the view); in addition, if data has changed, they can use their “cache” for data that hasn’t changed and use the base table for any data that has changed.

Regular views do not cache data, and therefore cannot improve performance by caching. However, in some cases, views help Snowflake generate a more efficient query plan. Also, both materialized views and regular views enhance data security by allowing data to be exposed or hidden at the row level or column level.

The following table shows key similarities and differences between tables, regular views, cached query results, and materialized views:

	Performance Benefits	Security Benefits	Simplifies Query Logic	Supports Clustering	Uses Storage	Uses Credits for Maintenance
Regular table				✔	✔	
Regular view		✔	✔			
Cached query result	✔					
Materialized view	✔	✔	✔	✔	✔	✔
External table						

Examples of Use Cases For Materialized Views

This section describes some general usage scenarios that also provide a conceptual overview of materialized views:

- Suppose that, every day, you run a query `Q` that includes a subquery `S`. If `S` is resource-intensive and queries data that changes only once a week, then you could improve performance of the outer query `Q` by running `S` and caching the results in a table named `CT`:

  - You would update the table only once a week.
  - The rest of the time, when you run `Q`, it would reference the subquery results of `S` that were stored in the table.

  This would work well as long as the results of subquery `S` change predictably (e.g. at the same time every week).

  However, if the results of `S` change unpredictably then caching the results in a table is risky; sometimes your main query `Q` will return out-of-date results if the results of subquery `S` are out of date (and thus the results of cached table `CT` are out of date).

  Ideally, you'd like a special type of cache for results that change rarely, but for which the timing of the change is unpredictable. Looking at it another way, you'd like to force your subquery `S` to be re-run (and your cache table `CT` to be updated) when necessary.

  A materialized view implements an approximation of the best of both worlds. You define a query for your materialized view, and the results of the query are cached (as though they were stored in an internal table), but Snowflake updates the cache when the table that the materialized view is defined on is updated. Thus, your subquery results are readily available for fast performance.

- As a less abstract example, suppose that you run a small branch of a large pharmacy, and your branch stocks hundreds of medications out of a total of tens of thousands of FDA-approved medications.

  Suppose also that you have a complete list of all medications that each of your customers takes, and that almost all of those customers order only medicines that are in stock (i.e. special orders are rare).

  In this scenario, you could create a materialized view that lists only the interactions among medicines that you keep in stock. When a customer orders a medicine that she has never used before, if both that medicine and all of the other medicines that she takes are covered by your materialized view, then you don't need to check the entire FDA database for drug interactions; you can just check the materialized view, so the search is faster.

- You can use a materialized view by itself, or you can use it in a join.

  Continuing with the pharmacy example, suppose that you have one table that lists all of the medicines that each of your customers takes; you can join that table to the materialized view of drug interactions to find out which of the customer's current medications might interact with the new medication.

  You might use an outer join to make sure that you list ***all*** of the customer's medicines, whether or not they are in your materialized view; if the outer join shows that any of the current medicines are not in the materialized view, you can re-run the query on the full drug interactions table.

How the Query Optimizer Uses Materialized Views

You don't need to specify a materialized view in a SQL statement in order for the view to be used. The query optimizer can automatically rewrite queries against the base table or regular views to use the materialized view instead.

For example, suppose that a materialized view contains all of the rows and columns that are needed by a query against a base table. The optimizer can decide to rewrite the query to use the materialized view, rather than the base table. This can dramatically speed up a query, especially if the base table contains a large amount of historical data.

As another example, in a multi-table join, the optimizer might decide to use a materialized view instead of a table for one of the tables in the join.

Note

Even if a materialized view can replace the base table in a particular query, the optimizer might not use the materialized view. For example, if the base table is clustered by a field, the optimizer might choose to scan the base table (rather than the materialized view) because the optimizer can effectively prune out partitions and provide equivalent performance using the base table.

A materialized view can also be used as the data source for a subquery.

When the optimizer chooses to use a materialized view implicitly, the materialized view is listed in the EXPLAIN plan or the Query Profile instead of the base table. You can use this information to experiment and understand which queries can benefit from existing materialized views.

About Materialized Views in Snowflake

The next sections explain how materialized views are represented in Snowflake.

DDL Commands For Materialized Views

Materialized views are first-class database objects. Snowflake provides the following DDL commands for creating and maintaining materialized views:

- CREATE MATERIALIZED VIEW
- ALTER MATERIALIZED VIEW
- DROP MATERIALIZED VIEW
- DESCRIBE MATERIALIZED VIEW
- SHOW MATERIALIZED VIEWS

DML Operations on Materialized Views

Snowflake does not allow standard DML (e.g. INSERT, UPDATE, DELETE) on materialized views. Snowflake does not allow users to truncate materialized views.

See Limitations on Working With Materialized Views (in this topic) for details.

Access Control Privileges

There are three types of privileges that are related to materialized views:

- Privileges on the schema that contains the materialized view.
- Privileges directly on the materialized view itself.
- Privileges on the database objects (e.g. tables) that the materialized view accesses.

You can use the standard commands for granting and revoking privileges on materialized views:

- GRANT <privileges>
- REVOKE <privileges>

Privileges on a Materialized View’s Schema

Materialized views consume storage space. To create a materialized view, you need the CREATE MATERIALIZED VIEW privilege on the schema that will contain the materialized view. You need to execute a statement similar to:

```
GRANT CREATE MATERIALIZED VIEW ON SCHEMA <schema_name> TO ROLE <role_name>
```

For more details about the GRANT statement, see GRANT <privileges>.

Privileges on a Materialized View

Materialized Views, like other database objects (tables, views, UDFs, etc.), are owned by a role and have privileges that can be granted to other roles.

You can grant the following privileges on a materialized view:

- SELECT

As with non-materialized views, a materialized view does not automatically inherit the privileges of its base table. You should explicitly grant privileges on the materialized view to the roles that should use that view.

Note
The exception to this rule is when the query optimizer rewrites a query against the base table to use the materialized view (as explained in How the Query Optimizer Uses Materialized Views). In this case, the user does not need privileges to use the materialized view in order to access the results of the query.

Privileges on the Database Objects Accessed by the Materialized View

As with non-materialized views, a user who wishes to access a materialized view needs privileges only on the view, not on the underlying object(s) that the view references.

Secure Materialized Views

Materialized views can be secure views.

Most information about secure views applies to secure materialized views. There are a few cases where secure materialized views are different from secure non-materialized views. The differences include:

- The command to find out whether a view is secure.
  - For non-materialized views, check the `IS_SECURE` column in the output of the `SHOW VIEWS` command.
  - For materialized views, check the `IS_SECURE` column in the output of the `SHOW MATERIALIZED VIEWS` command.

For more information about secure views, see Working with Secure Views.

The syntax to create secure materialized views is documented at CREATE MATERIALIZED VIEW.

Creating and Working With Materialized Views

This section provides information about creating and working with materialized views.

Planning to Create a Materialized View

When deciding to create a materialized view, consider doing some analysis to determine the need for the view:

1. Examine the filters, projections, and aggregations of queries that are frequent or expensive.
2. Use the Query Profile and the EXPLAIN command to see whether existing materialized views are already being used by the automatic query rewrite feature. You might find that you do not need to create any new materialized views if there are existing views that fit the queries well.
3. Before adding any materialized views, record current query costs and performance so that you can evaluate the difference after creating the new materialized view.
4. If you find queries with very selective filters that do not benefit from clustering the table, then a materialized view containing the same filters can help the queries avoid scanning a lot of data.

   Similarly, if you find queries that use aggregation, or that contain expressions that are very expensive to evaluate (for example, expensive function calls, or expensive operations on semi-structured data), then a materialized views that uses the same expression(s) or aggregation(s) can provide a benefit.

5. Run the EXPLAIN command against the original queries, or run the queries and check the Query Profile, to see whether the new materialized view is being used.
6. Monitor the combined query ***and*** materialized view costs, and evaluate whether the performance or cost benefits justify the cost of the materialized view's maintenance.

   Examine the query costs of the base table as well. In cases where the optimizer can rewrite the query to use a materialized view, query compilation can consume more time and resources. (The optimizer has a larger number of possibilities to consider.)
7. Remember that you can always reference materialized views directly if it simplifies your queries or you know that a materialized view will give you better performance. However, in most cases, you can simply query the base table and the automatic query rewrite feature will do that for you.

Creating a Materialized View

Use the CREATE MATERIALIZED VIEW command to create a materialized view. For an example, see Basic Example: Creating a Materialized View (in this topic).

Note the following:

- Whenever possible, use the fully-qualified name for the base table referenced in a materialized view. This insulates the view from changes that can invalidate the view, such as moving the base table to a different schema from the view (or vice versa).

  If the name of the base table is not qualified, and the table or view is moved to a different schema, the reference becomes invalid.
- If you are referring to the base table more than once in the view definition, use the same qualifier in all references to the base table. For example, if you choose to use the fully-qualified name, make sure that all references to the base table use the fully-qualified name.
- If you specify a filter when creating a materialized view (e.g. `WHERE column_1 BETWEEN Y and Z`), the optimizer can use the materialized view for queries against the base table that have the same filter or a more restrictive filter. Here are some examples:
  - Here's a simple example of range subsumption. In this example, the filter in the query does not match the filter in the materialized view. However, the filter in the query selects only rows that are in the materialized view, so the optimizer can choose to scan the materialized view rather than the entire table.

```
-- Example of a materialized view with a range filter
create materialized view v1 as
    select * from table1 where column_1 between 100 and 400;
```

```
-- Example of a query that might be rewritten to use the materializ
select * from table1 where column_1 between 200 and 300;
```

- This example shows OR subsumption. The materialized view contains all the rows that the subsequent query needs.

  Define a materialized view that contains all rows that have either value X or value Y:

  ```
  create materialized view mv1 as
      select * from tab1 where column_1 = X or column_1 = Y;
  ```

  Define a query that looks only for value Y (which is included in the materialized view):

  ```
  select * from tab1 where column_1 = Y;
  ```

  The query above can be rewritten internally as:

  ```
  select * from mv1 where column_1 = Y;
  ```

- This example is another example of OR subsumption. There's no explicit `OR` in the materialized view definition. However, an `IN` clause is equivalent to a series of `OR` expressions, so the optimizer can re-write this query the same way as it re-wrote the OR subsumption example above:

  ```
  create materialized view mv1 as
      select * from tab1 where column_1 in (X, Y);
  ```

Define a query that looks only for value Y (which is included in the materialized view):

```
select * from tab1 where column_1 = Y;
```

The query above can be rewritten internally as:

```
select * from mv1 where column_1 = Y;
```

- This example uses AND subsumption:

  Create a materialized view that contains all rows where `column_1 = X`.

  ```
  create materialized view mv2 as
      select * from table1 where column_1 = X;
  ```

  Create a query:

  ```
  select column_1, column_2 from table1 where column_1 = X AND columr
  ```

  The query can be rewritten as:

  ```
  select * from mv2 where column_2 = Y;
  ```

  The rewritten query does not even need to include the expression `column_1 = X` because the materialized view's definition already requires that all rows match `column_1 = X`.

- The following example shows aggregate subsumption:

  The materialized view is defined below:

  ```
  create materialized view mv4 as
      select column_1, column_2, sum(column_3) from table1 group by c
  ```

The following query can use the materialized view defined above:

```
select column_1, sum(column_3) from table1 group by column_1;
```

The query can be rewritten as:

```
select column_1, sum(column_3) from mv4 group by column_1;
```

The rewritten query does not take advantage of the additional grouping by column_2, but the rewritten query is not blocked by that additional grouping, either.

- The CREATE MATERIALIZED VIEW statement might take a substantial amount of time to complete.

  When a materialized view is first created, Snowflake performs the equivalent of a CTAS (CREATE TABLE … AS ….) operation.

Limitations on Creating Materialized Views

Note
These are current limitations; some of them might be removed or changed in future versions.

The following limitations apply to creating materialized views:

- A materialized view can query only a single table.
- Joins, including self-joins, are not supported.
- A materialized view cannot query:
  - A materialized view.
  - A non-materialized view.
  - A UDTF (user-defined table function).
- A materialized view cannot include:
  - UDFs (this limitation applies to all types of user-defined functions, including external functions).

  - Window functions.
  - HAVING clauses.
  - ORDER BY clause.
  - LIMIT clause.
  - GROUP BY keys that are not within the SELECT list. All GROUP BY keys in a materialized view must be part of the SELECT list.
  - GROUP BY GROUPING SETS.
  - GROUP BY ROLLUP.
  - GROUP BY CUBE.
  - Nesting of subqueries within a materialized view.
  - The MINUS, EXCEPT, or INTERSECT set operators.
- Many aggregate functions are not allowed in a materialized view definition.
  - The aggregate functions that are ***supported*** in materialized views are:
    - APPROX_COUNT_DISTINCT (HLL).
    - AVG (except when used in PIVOT).
    - BITAND_AGG.
    - BITOR_AGG.
    - BITXOR_AGG.
    - COUNT.
    - MIN.
    - MAX.
    - STDDEV.
    - STDDEV_POP.
    - STDDEV_SAMP.
    - SUM.
    - VARIANCE (VARIANCE_SAMP, VAR_SAMP).
    - VARIANCE_POP (VAR_POP).

    The other aggregate functions are ***not supported*** in materialized views.

Note

Aggregate functions that are allowed in materialized views still have some restrictions:

- Aggregate functions cannot be nested.
- Aggregate functions used in complex expressions (e.g. `(sum(salary)/10)`) can only be used in the outer-most level of a query, not in a subquery or an in-line view.

  For example, the following is allowed:

  ```
  create materialized view mv1 as
      select
          sum(x) + 100
        from t;
  ```

  The following is ***not*** allowed:

  ```
  create materialized view mv2 as
      select
          y + 10
        from (
          select
            sum(x) as y
          from t
        );
  ```

- DISTINCT cannot be combined with aggregate functions.
- In a materialized view, the aggregate functions `AVG`, `COUNT`, `MIN`, `MAX`, and `SUM` can be used as aggregate functions but not as window functions. In a materialized view, these functions cannot be used with the `OVER` clause:

  ```
  OVER ( [ PARTITION BY <expr1> ] [ ORDER BY <expr2> ] )
  ```

- If an aggregate function is in a subquery, the materialized view cannot create an expression on top of the aggregated column(s) from that subquery. For example, consider the following materialized view definition:

```
create or replace materialized view mv1 as
    select c1 + 10 as c1new, c2
        from (select sum(c1) as c1, c2 from t group by c2);
```

The expression “c1 + 10” is an expression on top of an aggregate function in a subquery, and therefore causes an error message.

Note that even an equality operator counts as an expression, which means that `CASE` expressions using columns that represent aggregate functions in a subquery are also prohibited.

To work around this limitation, create a materialized view without the expression, and then create a non-materialized view that includes the expression, for example:

```
create or replace materialized view mv1 as
    select c1, c2
        from (select sum(c1) as c1, c2 from t group by c2);

create or replace view expr_v1 as
    select c1 + 10 as c1new, c2
        from (select c1, c2 from mv1);
```

- Functions used in a materialized view must be deterministic. For example, using CURRENT_TIME or CURRENT_TIMESTAMP is not permitted.
- A materialized view should not be defined using a function that produces different results for different settings of parameters, such as the session-level parameter TIMESTAMP_TYPE_MAPPING.

  For example, suppose that a view is defined as follows:

  ```
  create materialized view bad_example (ts1) as
      select to_timestamp(n) from t1;
  ```

  The data type of the return value from `TO_TIMESTAMP(n)` depends upon the parameter TIMESTAMP_TYPE_MAPPING, so the contents of the materialized view depend upon the value of TIMESTAMP_TYPE_MAPPING at the time that the view was created.

When a materialized view is created, the expression defining each of its columns is evaluated and stored. If a column definition depends upon a particular session variable, and the session variable changes, the expression is not re-evaluated, and the materialized view is not updated. If the materialized view depends upon a particular value of a session variable, and if the session variable's value has changed, then queries on the materialized view fail.

To avoid this problem, force the expression to a value that does not depend upon any session variables. The example below casts the output to a particular data type, independent of the TIMESTAMP_TYPE_MAPPING parameter:

```
create materialized view good_example (ts1) as
    select to_timestamp(n)::TIMESTAMP_NTZ from t1;
```

This issue is specific to ***materialized*** views. Non-materialized views generate their output dynamically based on current parameter settings, so the results can't be stale.

- Materialized views cannot be created using the Time Travel feature.

Basic Example: Creating a Materialized View

This section contains a basic example of creating and using a materialized view:

```
CREATE OR REPLACE MATERIALIZED VIEW mv1 AS
  SELECT My_ResourceIntensive_Function(binary_col) FROM table1;

SELECT * FROM mv1;
```

More detailed examples are provided in Examples (in this topic).

Understanding How Materialized Views Are Maintained

After you create a materialized view, a background process automatically maintains the data in the materialized view. Note the following:

- Maintenance of materialized views is performed by a background process, and the timing is optimally based on the workload on the base table and the materialized view.

    - This process updates the materialized view with changes made by DML operations to the base table (insertions, updates, and deletions).

      In addition, clustering on the base table can also result in refreshes of a materialized view. Refer to Best Practices for Clustering Materialized Views and their Base Tables.

    - When rows are inserted in the base table, the process performs a “refresh” operation to insert the new rows into the materialized view.
    - When rows are deleted in the base table, the process performs a “compaction” operation on the materialized view, deleting these rows from the materialized view.

- To see the last time that a materialized view was refreshed, execute the SHOW MATERIALIZED VIEWS command.

  Check the REFRESHED_ON and BEHIND_BY columns in the output:

    - The REFRESHED_ON and COMPACTED_ON columns show the timestamp of the last DML operation on the base table that was processed by the refresh and compaction operations, respectively.
    - The BEHIND_BY column indicates the amount of time that the updates to the materialized view are behind the updates to the base table.

- If maintenance falls behind, queries might run more slowly than when the views are up-to-date, but the results will always be up-to-date.

  If some micro-partitions of the materialized view are out of date, Snowflake skips those partitions and looks up the data from the base table.

Suspending and Resuming Maintenance on a Materialized View

If you need to suspend the maintenance and use of a materialized view, execute the ALTER MATERIALIZED VIEW command with the SUSPEND parameter:

```
ALTER MATERIALIZED VIEW <name> SUSPEND
```

If you suspend maintenance of a view, you cannot query the view until you resume maintenance.

To resume the maintenance and use of a materialized view, execute the ALTER MATERIALIZED VIEW command with the RESUME parameter:

```
ALTER MATERIALIZED VIEW <name> RESUME
```

For an example, see Suspending Updates to a Materialized View.

Displaying Information About Materialized Views

The following command and view provide information about materialized views:

- The SHOW VIEWS command returns information about both materialized and regular views.
- The INFORMATION_SCHEMA.TABLES view shows materialized views. The `TABLE_TYPE` column shows "MATERIALIZED VIEW". The `IS_INSERTABLE` column is always "NO", because you cannot insert directly into a materialized view.

  Note
  The INFORMATION_SCHEMA.VIEWS view does not show materialized views. Materialized views are shown by INFORMATION_SCHEMA.TABLES.

Limitations on Working With Materialized Views

Note
These are current limitations; some of them might be removed or changed in future versions.

The following limitations apply to using materialized views:

- To ensure that materialized views stay consistent with the base table on which they are defined, you cannot perform most DML operations on a materialized view itself. For example, you cannot insert rows directly into a materialized view (although of course you can insert rows into the base table). The prohibited DML operations include:
  - COPY
  - DELETE
  - INSERT
  - MERGE

- UPDATE

  Truncating a materialized view is not supported.

- You cannot directly clone a materialized view by using the `CREATE MATERIALIZED VIEW ... CLONE...` command. However, if you clone a schema or a database that contains a materialized view, the materialized view will be cloned and included in the new schema or database.
- Snowflake does not support using the Time Travel feature to query materialized views at a point in the past (e.g. using the AT clause when querying a materialized view).

  However, you can use Time Travel to clone a database or schema containing a materialized view at a point in the past. For details, see Materialized Views and Time Travel.

- Materialized Views are not monitored by Snowflake Working with Resource Monitors.

Effects of Changes to Base Tables on Materialized Views

The following sections explain how materialized views are affected by changes to the base tables.

Adding Columns to the Base Table

If columns are added to the base table, those new columns are ***not*** propagated to the materialized view automatically.

This is true even if the materialized view was defined with `SELECT *` (e.g. `CREATE MATERIALIZED VIEW AS SELECT * FROM table2 ...`). The columns of the materialized view are defined at the time that the materialized view is defined. The `SELECT *` is not interpreted dynamically each time that the materialized view is queried.

To avoid confusion, Snowflake recommends not using `SELECT *` in the definition of a materialized view.

Note

Adding a column to the base table does not suspend a materialized view created on that base table.

Changing or Dropping Columns in the Base Table

If a base table is altered so that existing columns are changed or dropped, then all materialized views on that base table are suspended; the materialized views cannot be used or maintained. (This is true even if the modified or dropped column was not part of the materialized view.)

You cannot RESUME that materialized view. If you want to use it again, you must recreate it.

The simplest way to recreate a materialized view with the same privileges on the view is by running the command:

```
CREATE OR REPLACE MATERIALIZED VIEW <view_name> ... COPY GRANTS ...
```

This is more efficient than running separate commands to:

1. Drop the materialized view (DROP MATERIALIZED VIEW).
2. Create the materialized view again (CREATE MATERIALIZED VIEW).
3. Create the same privileges on the view (GRANT and REVOKE).

Renaming or Swapping the Base Table

Renaming or swapping the base table (or the schema or database containing the base table) can result in the materialized view pointing to a different base table than the base table used to create the materialized view. The following are examples of situations in which this can occur:

- The base table is renamed (through ALTER TABLE … RENAME), and another table is created with the original name of the base table.
- The base table of a materialized view is swapped with another table (through ALTER TABLE … SWAP WITH).
- The schema or database containing the base table of the materialized view is moved through DROP, SWAP or RENAME.

In these cases, the materialized view is suspended. In most cases, you must recreate the materialized view in order to use the view.

Dropping the Base Table

If a base table is dropped, the materialized view is suspended (but not automatically dropped).

In most cases, the materialized view must be dropped.

If for some reason you are recreating the base table and would also like to recreate the materialized view with the same definition it had previously, then first recreate the base table and then replace the view by using `CREATE OR REPLACE MATERIALIZED VIEW <view_name> ... COPY GRANTS ...`.

Materialized Views in Cloned Schemas and Databases

If you clone a schema or a database that contains a materialized view, then the materialized view is cloned.

If you clone the materialized view and the corresponding base table at the same time (as part of the same `CREATE SCHEMA ... CLONE` or `CREATE DATABASE ... CLONE` operation), then the cloned materialized view refers to the cloned base table.

If you clone the materialized view without cloning the base table (e.g. if the table is in Database1.Schema1 and the view is in Database1.Schema2, and you clone only Schema2 rather than all of Database1), then the cloned view will refer to the original base table.

Materialized Views Cost

Materialized views impact your costs for both storage and compute resources:

- Storage: Each materialized view stores query results, which adds to the monthly storage usage for your account.
- Compute resources: In order to prevent materialized views from becoming out-of-date, Snowflake performs automatic background maintenance of materialized views. When a base table changes, all materialized views defined on the table are updated by a background service that uses compute resources provided by Snowflake.

  These updates can consume significant resources, resulting in increased credit usage. However, Snowflake ensures efficient credit usage by billing your account only for the actual resources used. Billing is calculated in 1-second increments.

To learn how many credits per compute-hour are consumed by materialized views, refer to the “Serverless Feature Credit Table” in the Snowflake service consumption table.

Estimating and Controlling Costs

There are no tools to estimate the costs of maintaining materialized views. In general, the costs are proportional to:

- The number of materialized views created on each base table, and the amount of data that changes in each of those materialized views when the base table changes. Any changes to micro-partitions in the base table require eventual materialized view maintenance, whether those changes are due to reclustering or DML statements run on the base table.
- The number of those materialized views that are clustered. Maintaining clustering (of either a table or a materialized view) adds costs.

  If a materialized view is clustered differently from the base table, the number of micro-partitions changed in the materialized view might be substantially larger than the number of micro-partitions changed in the base table.

  For example, consider the case where the base table is changed largely by inserting (appending) data, and is not clustered, so the base table is largely in the order that the rows were inserted into the table. Imagine that the materialized view is clustered by an independent column, for example, postal code. If 100 new rows are added to the base table, those might go into one or two new micro-partitions, leaving the other micro-partitions in the base table untouched. But those 100 rows might require re-writing 100 micro-partitions in the clustered materialized view.

  As another example, consider deletes. Deleting the oldest rows in an unclustered base table might delete only the oldest micro-partitions, but might require changes to a far larger number of micro-partitions in a materialized view that is not clustered by age.

  (For more details about clustering materialized views, see Materialized Views and Clustering.)

You can control the cost of maintaining materialized views by carefully choosing how many views to create, which tables to create them on, and each view’s definition (including the number of rows and columns in that view).

You can also control costs by suspending or resuming the materialized view; however, suspending maintenance typically only defers costs, rather than reducing them. The longer that maintenance has been deferred, the more maintenance there is to do.

See also Best Practices for Maintaining Materialized Views.

Tip
If you are concerned about the cost of maintaining materialized views, Snowflake recommends starting slowly with this feature (i.e. create only a few materialized views on selected tables) and monitor the costs over time.

Viewing Costs

You can view the billing costs for maintaining materialized views using Snowsight, the Classic Console, or SQL:

Snowsight As a user with the proper privileges, click on **Admin** » **Usage**.

Classic Console As an account administrator, click on **Account** » **Billing & Usage**.

The credit costs are tracked in a Snowflake-provided virtual warehouse named **MATERIALIZED_VIEW_MAINTENANCE**.

SQL Query either of the following:

- MATERIALIZED_VIEW_REFRESH_HISTORY table function (in the Snowflake Information Schema).

  For example:

```
SELECT * FROM TABLE(INFORMATION_SCHEMA.MATERIALIZED_VI
```

- MATERIALIZED_VIEW_REFRESH_HISTORY View view (in Account Usage).

  The following queries can be executed against the MATERIALIZED_VIEW_REFRESH_HISTORY view:

  Query: Materialized Views cost history (by day, by object)

This query provides a full list of materialized views and the volume of credits consumed via the service over the last 30 days, broken out by day. Any irregularities in the credit consumption or consistently high consumption are flags for additional investigation.

```
SELECT TO_DATE(start_time) AS date,
  database_name,
  schema_name,
  table_name,
  SUM(credits_used) AS credits_used
FROM snowflake.account_usage.materialized_view_refresh
WHERE start_time >= DATEADD(month,-1,CURRENT_TIMESTAMP
GROUP BY 1,2,3,4
ORDER BY 5 DESC;
```

Query: Materialized Views History & m-day average

This query shows the average daily credits consumed by materialized views grouped by week over the last year. It can help identify anomalies in daily averages over the year so you can investigate spikes or unexpected changes in consumption.

```
WITH credits_by_day AS (
  SELECT TO_DATE(start_time) AS date,
    SUM(credits_used) AS credits_used
  FROM snowflake.account_usage.materialized_view_refre
  WHERE start_time >= DATEADD(year,-1,CURRENT_TIMESTAM
  GROUP BY 1
  ORDER BY 2 DESC
)

SELECT DATE_TRUNC('week',date),
  AVG(credits_used) AS avg_daily_credits
FROM credits_by_day
GROUP BY 1
ORDER BY 1;
```

Note

Resource monitors provide control over virtual warehouse credit usage; however, you cannot use them to control credit usage for the Snowflake-provided warehouses, including the **MATERIALIZED_VIEW_MAINTENANCE** warehouse.

Materialized Views and Clustering

Defining a clustering key on a materialized view is supported and can increase performance in many situations. However, it also adds costs.

If you cluster both the materialized view(s) and the base table on which the materialized view(s) are defined, you can cluster the materialized view(s) on different columns from the columns used to cluster the base table.

In most cases, clustering a subset of the materialized views on a table tends to be more cost-effective than clustering the table itself. If the data in the base table is accessed (almost) exclusively through the materialized views, and (almost) never directly through the base table, then clustering the base table adds costs without adding benefit.

If you are considering clustering both the base table and the materialized views, Snowflake recommends that you start by clustering only the materialized views, and that you monitor performance and cost before and after adding clustering to the base table.

If you plan to create a table, load it, and create a clustered materialized view(s) on the table, then Snowflake recommends that you create the materialized views last (after loading as much data as possible). This can save money on the initial data load, because it avoids some extra effort to maintain the clustering of the materialized view the first time that the materialized view is loaded.

For more details about clustering, refer to:

- Understanding Snowflake Table Structures
- Automatic Clustering

For more information about the costs of clustering materialized views, refer to:

- Materialized Views Cost
- Best Practices for Materialized Views

Materialized Views and Time Travel

Currently, you cannot use Time Travel to query historical data for materialized views.

However, note the following:

- You can use Time Travel to clone a database or schema containing a materialized view at a specific point in the past. Snowflake clones the materialized view at the specified point in time.
- To support cloning with Time Travel, Snowflake does maintain historical data for materialized views. You will be billed for the storage costs for historical data for materialized views.
- The storage costs depend on the data retention period for the materialized views, which is determined by the DATA_RETENTION_TIME_IN_DAYS parameter. Materialized views inherit this parameter from its parent schema or database.

Best Practices for Materialized Views

The following sections summarize the best practices for working with materialized views:

- Best Practices for Creating Materialized Views
- Best Practices for Maintaining Materialized Views
- Best Practices for Clustering Materialized Views and their Base Tables

Best Practices for Creating Materialized Views

- Most materialized views should do one or both of the following:
  - Filter data. You can do this by:
    - Filtering rows (e.g. defining the materialized view so that only very recent data is included). In some applications, the best data to store is the abnormal data. For example, if you are monitoring pressure in a gas pipeline to estimate when pipes might fail, you might store all pressure data in the base table, and store only unusually high pressure measurements in the materialized view. Similarly, if you are monitoring network traffic, your base table might store all monitoring information, while your materialized view might store only unusual and suspicious information (e.g. from IP addresses known to launch DOS (Denial Of Service) attacks).
    - Filtering columns (e.g. selecting specific columns rather than “SELECT * …”). Using `SELECT * ...` to define a materialized view typically is expensive. It can also lead to future errors; if columns are added to the base table later (e.g. `ALTER TABLE ... ADD COLUMN ...`), the materialized view does not automatically incorporate the new columns.
  - Perform resource-intensive operations and store the results so that the resource intensive operations don’t need to be performed as often.

- You can create more than one materialized view for the same base table. For example, you can create one materialized view that contains just the most recent data, and another materialized view that stores unusual data. You can then create a non-materialized view that joins the two tables and shows recent data that matches unusual historical data so that you can quickly detect unusual situations, such as a DOS (denial of service) attack that is ramping up.

  Snowflake recommends materialized views for unusual data only when:

  - The base table is not clustered, or the columns that contain the unusual data are not already part of the base table's clustering key.
  - The data is unusual enough that it is easy to isolate, but not so unusual that it is rarely used. (If the data is rarely used, the cost of maintaining the materialized view is likely to outweigh the performance benefit and cost savings from being able to access it quickly when it is used.)

Best Practices for Maintaining Materialized Views

- Snowflake recommends batching DML operations on the base table:
  - `DELETE`: If tables store data for the most recent time period (e.g. the most recent day or week or month), then when you trim your base table by deleting old data, the changes to the base table are propagated to the materialized view. Depending upon how the data is distributed across the micro-partitions, this could cause you to pay more for background updates of the materialized views. In some cases, you might be able to reduce costs by deleting less frequently (e.g. daily rather than hourly, or hourly rather than every 10 minutes).

    If you do not need to keep a specific amount of old data, you should experiment to find the best balance between cost and functionality.
  - `INSERT`, `UPDATE`, and `MERGE`: Batching these types of DML statements on the base table can reduce the cost of maintaining the materialized views.

Best Practices for Clustering Materialized Views and their Base Tables

- If you create a materialized view on a base table, and if the materialized views are accessed frequently and the base table is not accessed frequently, it is usually more efficient to avoid clustering the base table.

If you create a materialized view on a clustered table, consider removing any clustering on the base table, because any change to the clustering of the base table will eventually require a refresh of the materialized view, which adds to the materialized view's maintenance costs.

- Clustering materialized views, especially materialized views on base tables that change frequently, increases costs. Do not cluster more materialized views than you need to.
- Almost all information about clustering tables also applies to clustering materialized views. For more information about clustering tables, see Strategies for Selecting Clustering Keys.

Examples

This section contains additional examples of creating and using materialized views. For a simple, introductory example, see Basic Example: Creating a Materialized View (in this topic).

Simple Materialized View

This first example illustrates a simple materialized view and a simple query on the view.

Create the table and load the data, and create the view:

```
CREATE TABLE inventory (product_ID INTEGER, wholesale_price FLOAT,
  description VARCHAR);

CREATE OR REPLACE MATERIALIZED VIEW mv1 AS
  SELECT product_ID, wholesale_price FROM inventory;

INSERT INTO inventory (product_ID, wholesale_price, description) VALUES
    (1, 1.00, 'cog');
```

Select data from the view:

```
SELECT product_ID, wholesale_price FROM mv1;
+------------+-----------------+
| PRODUCT_ID | WHOLESALE_PRICE |
|------------+-----------------|
|          1 |                1 |
+------------+-----------------+
```

Joining a Materialized View

You can join a materialized view with a table or another view. This example builds on the previous example by creating an additional table, and then a non-materialized view that shows profits by joining the materialized view to a table:

```
CREATE TABLE sales (product_ID INTEGER, quantity INTEGER, price FLOAT);

INSERT INTO sales (product_ID, quantity, price) VALUES
   (1,  1, 1.99);

CREATE or replace VIEW profits AS
  SELECT m.product_ID, SUM(IFNULL(s.quantity, 0)) AS quantity,
      SUM(IFNULL(quantity * (s.price - m.wholesale_price), 0)) AS profit
    FROM mv1 AS m LEFT OUTER JOIN sales AS s ON s.product_ID = m.product
    GROUP BY m.product_ID;
```

Select data from the view:

```
SELECT * FROM profits;
+------------+----------+--------+
| PRODUCT_ID | QUANTITY | PROFIT |
|------------+----------+--------|
|          1 |        1 |   0.99 |
+------------+----------+--------+
```

Suspending Updates to a Materialized View

The following example temporarily suspends the use (and maintenance) of the `mv1` materialized view, and shows that queries on that view generate an error message while the materialized view is suspended:

```
ALTER MATERIALIZED VIEW mv1 SUSPEND;

INSERT INTO inventory (product_ID, wholesale_price, description) VALUES
```

```
    (2, 2.00, 'sprocket');

INSERT INTO sales (product_ID, quantity, price) VALUES
   (2, 10, 2.99),
   (2,  1, 2.99);
```

Select data from the materialized view:

```
SELECT * FROM profits ORDER BY product_ID;
```

Output:

```
002037 (42601): SQL compilation error:
Failure during expansion of view 'PROFITS': SQL compilation error:
Failure during expansion of view 'MV1': SQL compilation error: Materiali
```

Resume:

```
ALTER MATERIALIZED VIEW mv1 RESUME;
```

Select data from the materialized view:

```
SELECT * FROM profits ORDER BY product_ID;
+------------+----------+--------+
| PRODUCT_ID | QUANTITY | PROFIT |
|------------+----------+--------|
|          1 |        1 |   0.99 |
|          2 |       11 |  10.89 |
+------------+----------+--------+
```

Clustering a Materialized View

This example creates a materialized view and then later clusters it:

These statements create two tables that track information about segments of a pipeline (e.g. for natural gas).

The segments that are most likely to fail in the near future are often the segments that are oldest, or that are made of materials that corrode easily, or that had experienced periods of unusually high pressure, so this example tracks each pipe’s age, pressure, and material (iron, copper, PVC plastic, etc.).

```
CREATE TABLE pipeline_segments (
    segment_ID BIGINT,
    material VARCHAR, -- e.g. copper, cast iron, PVC.
    installation_year DATE,  -- older pipes are more likely to be corr
    rated_pressure FLOAT  -- maximum recommended pressure at installat
    );

INSERT INTO pipeline_segments
    (segment_ID, material, installation_year, rated_pressure)
  VALUES
    (1, 'PVC', '1994-01-01'::DATE, 60),
    (2, 'cast iron', '1950-01-01'::DATE, 120)
    ;

CREATE TABLE pipeline_pressures (
    segment_ID BIGINT,
    pressure_psi FLOAT,  -- pressure in Pounds per Square Inch
    measurement_timestamp TIMESTAMP
    );
INSERT INTO pipeline_pressures
   (segment_ID, pressure_psi, measurement_timestamp)
  VALUES
    (2, 10, '2018-09-01 00:01:00'),
    (2, 95, '2018-09-01 00:02:00')
    ;
```

The pipeline segments don’t change very frequently, and the oldest pipeline segments are the segments most likely to fail, so create a materialized view of the oldest segments.

```
CREATE MATERIALIZED VIEW vulnerable_pipes
  (segment_ID, installation_year, rated_pressure)
  AS
    SELECT segment_ID, installation_year, rated_pressure
        FROM pipeline_segments
        WHERE material = 'cast iron' AND installation_year < '1980'::D
```

You can add clustering or change the clustering key. For example, to cluster on `installation_year`:

```
ALTER MATERIALIZED VIEW vulnerable_pipes CLUSTER BY (installation_year
```

New pressure measurements arrive frequently (perhaps every 10 seconds), so maintaining a materialized view on the pressure measurements would be expensive. Therefore, even though high performance (fast retrieval) of recent pressure data is important, the `pipeline_pressures` table starts without a materialized view.

If performance is too slow, you can create a materialized view that contains only recent pressure data, or that contains data only about abnormal high-pressure events.

Create a (non-materialized) view that combines information from the materialized view and the `pipeline_pressures` table:

```
CREATE VIEW high_risk AS
    SELECT seg.segment_ID, installation_year, measurement_timestamp::DA
         DATEDIFF('YEAR', installation_year::DATE, measurement_timestar
         rated_pressure - age AS safe_pressure, pressure_psi AS actual.
      FROM vulnerable_pipes AS seg INNER JOIN pipeline_pressures AS ps
         ON psi.segment_ID = seg.segment_ID
      WHERE pressure_psi > safe_pressure
      ;
```

Now list the high-risk pipeline segments:

```
SELECT * FROM high_risk;
+------------+-------------------+------------------+-----+---------
| SEGMENT_ID | INSTALLATION_YEAR | MEASUREMENT_DATE | AGE | SAFE_PRES
|------------+-------------------+------------------+-----+---------
|          2 | 1950-01-01        | 2018-09-01       |  68 |
+------------+-------------------+------------------+-----+---------
```

This shows that the pipeline segment with `segment_id = 2`, which is made of a material that corrodes, is old. This segment has never experienced pressure higher than the maximum pressure rating at the time it was installed, but because of the potential for corrosion, its “safe limit” has declined over time, and the highest pressure it has experienced is higher than the pressure that was recommended for a pipe as old as the pipe was at the time of the pressure measurement.

Creating a Materialized View on Shared Data

You can create a materialized view on shared data.

Account1:

```
create or replace table db1.schema1.table1(c1 int);
create or replace share sh1;
grant usage on database db1 to share sh1;
alter share sh1 add accounts = account2;
grant usage on schema db1.schema1 to share sh1;
grant select on table db1.schema1.table1 to share sh1;
```

Account2:

```
create or replace database dbshared from share account1.sh1;
create or replace materialized view mv1 as select * from dbshared.schema1.
```

Note
Remember that maintaining materialized views will consume credits. When you create a materialized view on someone else's shared table, the changes to that shared table will result in charges to you as your materialized view is maintained.

Sharing a Materialized View

You can use Snowflake's data sharing feature to share a materialized view.

For more information about data sharing, see Overview of Data Sharing at Snowflake.

Note
Remember that maintaining materialized views will consume credits. When someone else creates a materialized view on your shared data, any changes to your shared data can cause charges to the people who have materialized views on your shared data. The larger the number of materialized views on a shared base table, the more important it is to update that base table efficiently to minimize the costs of maintaining materialized views.

Troubleshooting

Compilation Error: `Failure during expansion of view '<name>': SQL compilation error: Materialized View <name> is invalid.`

Possible Causes

- This can be caused by trying to use a suspended view. For more information about suspending and resuming views, see ALTER MATERIALIZED VIEW.
- This can be caused by a change to the base table that underlies the materialized view. For example, this error is returned if:
  - The base table is dropped.
  - The materialized view refers to a base table column that has been dropped.

Possible Solutions

- If the view has been suspended:
  - Consider resuming the view by executing ALTER MATERIALIZED VIEW … RESUME.
  - Consider running the query against the base table. However, this is likely to consume more credits and take longer than running the query against the materialized view.
- If the base table has been modified or dropped:
  - If the base table has been dropped, then drop the materialized view.
  - If the base table has been modified (e.g. has dropped a column referenced by the view), and if the materialized view would still be useful with the new version of the table, then consider dropping and re-creating the materialized view, using the columns that remain in the base table.
  - If no other cause of the error message is apparent, consider dropping and re-creating the materialized view.
  - Consider running the query against the base table. However, this is likely to consume more credits and take longer than running the query against the materialized view.

Note

If the materialized view is ***invalid***, Snowflake does not attempt to substitute for the materialized view by reading the data from the underlying table.

This contrasts with the behavior when the materialized view is ***valid but not up-to-date*** (e.g. when the materialized view's background maintenance process is still running). When the view is valid but not up-to-date, Snowflake reads from the base table, as described in Understanding How Materialized Views Are Maintained.

SHOW MATERIALIZED VIEWS Command Shows Materialized Views that are Not Updated

Possible Cause

One possible cause is that the refresh failed because the SELECT in the view definition failed.

If the `SELECT` fails during the refresh, then the refresh will fail; however, because the refresh is done behind the scenes, the user will not see an error message at the time the refresh is attempted.

If some or all of the data in the materialized view is out of date, then Snowflake will retrieve up-to-date data from the base table; Snowflake will not issue an error message that the materialized view was not refreshed.

Therefore, neither the refresh nor subsequent queries necessarily shows that the `SELECT` in the view failed. To detect whether refreshes are failing, use the command SHOW MATERIALIZED VIEWS and look for the column named `behind_by`. If the data is not getting refreshed, then the `SELECT` might be failing.

Possible Solution

Make sure that the underlying table exists. If you drop the table that the view is defined on, but you don't drop the view, then the view will continue to exist.

In some cases, you might be able to debug the problem by manually running the `SELECT` statement in the materialized view's definition, or by running a simpler (less expensive) `SELECT` on the table referenced in the materialized view's definition.

If you do not know the exact definition of the materialized view, you can find it in the output of SHOW MATERIALIZED VIEWS or by using the GET_DDL function.