Micro-partitions & Data Clustering

https://docs.snowflake.net/manuals/user-guide/tables-clustering-micropartitions.html

Traditional data warehouses rely on static partitioning of large tables to achieve acceptable performance and enable better scaling. In these systems, a *partition* is a unit of management that is manipulated independently using specialized DDL and syntax; however, static partitioning has a number of well-known limitations, such as maintenance overhead and data skew, which can result in disproportionately-sized partitions.

Snowflake has implemented a powerful and unique form of partitioning, called *micro-partitioning*, that delivers all the advantages of static partitioning without the known limitations, as well as providing additional significant benefits.

**Sections in this topic:**

* [What are Micro-partitions?](https://docs.snowflake.net/manuals/user-guide/tables-clustering-micropartitions.html#what-are-micro-partitions)
* [Benefits of Micro-partitioning](https://docs.snowflake.net/manuals/user-guide/tables-clustering-micropartitions.html#benefits-of-micro-partitioning)
* [Impact of Micro-partitions](https://docs.snowflake.net/manuals/user-guide/tables-clustering-micropartitions.html#impact-of-micro-partitions)
  + [DML](https://docs.snowflake.net/manuals/user-guide/tables-clustering-micropartitions.html#dml)
  + [Query Pruning](https://docs.snowflake.net/manuals/user-guide/tables-clustering-micropartitions.html#query-pruning)
* [What is Data Clustering?](https://docs.snowflake.net/manuals/user-guide/tables-clustering-micropartitions.html#what-is-data-clustering)
* [Clustering Information Maintained for Micro-partitions](https://docs.snowflake.net/manuals/user-guide/tables-clustering-micropartitions.html#clustering-information-maintained-for-micro-partitions)
  + [Clustering Depth](https://docs.snowflake.net/manuals/user-guide/tables-clustering-micropartitions.html#clustering-depth)
  + [Clustering Depth Illustrated](https://docs.snowflake.net/manuals/user-guide/tables-clustering-micropartitions.html#clustering-depth-illustrated)
* [Monitoring Clustering Information for Tables](https://docs.snowflake.net/manuals/user-guide/tables-clustering-micropartitions.html#monitoring-clustering-information-for-tables)

What are Micro-partitions?

All data in Snowflake tables is automatically divided into micro-partitions, which are contiguous units of storage. Each micro-partition contains between 50 MB and 500 MB of uncompressed data (note that the actual size in Snowflake is smaller because data is always stored compressed). Groups of rows in tables are mapped into individual micro-partitions, organized in a columnar fashion. This size and structure allows for extremely granular pruning of very large tables, which can be comprised of millions, or even hundreds of millions, of micro-partitions.

Snowflake stores metadata about all rows stored in a micro-partition, including:

* The range of values for each of the columns in the micro-partition.
* The number of distinct values.
* Additional properties used for both optimization and efficient query processing.

**Note**

Micro-partitioning is automatically performed on all Snowflake tables. Tables are transparently partitioned using the ordering of the data as it is inserted/loaded.

Benefits of Micro-partitioning

The benefits of Snowflake’s approach to partitioning table data include:

* In contrast to traditional static partitioning, Snowflake micro-partitions are derived automatically; they don’t need to be explicitly defined up-front or maintained by users.
* As the name suggests, micro-partitions are small in size (50 to 500 MB, before compression), which enables extremely efficient DML and fine-grained pruning for faster queries.
* Micro-partitions can overlap in their range of values, which, combined with their uniformly small size, helps prevent skew.
* Columns are stored independently within micro-partitions, often referred to as *columnar storage*. This enables efficient scanning of individual columns; only the columns referenced by a query are scanned.
* Columns are also compressed individually within micro-partitions. Snowflake automatically determines the most efficient compression algorithm for the columns in each micro-partition.

Impact of Micro-partitions

DML

All DML operations (e.g. DELETE, UPDATE, MERGE) take advantage of the underlying micro-partition metadata to facilitate and simplify table maintenance. For example, some operations, such as deleting all rows from a table, are metadata-only operations.

Query Pruning

The micro-partition metadata maintained by Snowflake enables precise pruning of columns in micro-partitions at query run-time, including columns containing semi-structured data. In other words, a query that specifies a filter predicate on a range of values that accesses 10% of the values in the range should ideally only scan 10% of the micro-partitions.

For example, assume a large table contains one year of historical data with date and hour columns. Assuming uniform distribution of the data, a query targeting a particular hour would ideally scan 1/8760th of the micro-partitions in the table and then only scan the portion of the micro-partitions that contain the data for the hour column; Snowflake uses columnar scanning of partitions so that an entire partition is not scanned if a query only filters by one column.

In other words, the closer the ratio of scanned micro-partitions and columnar data is to the ratio of actual data selected, the more efficient is the pruning performed on the table.

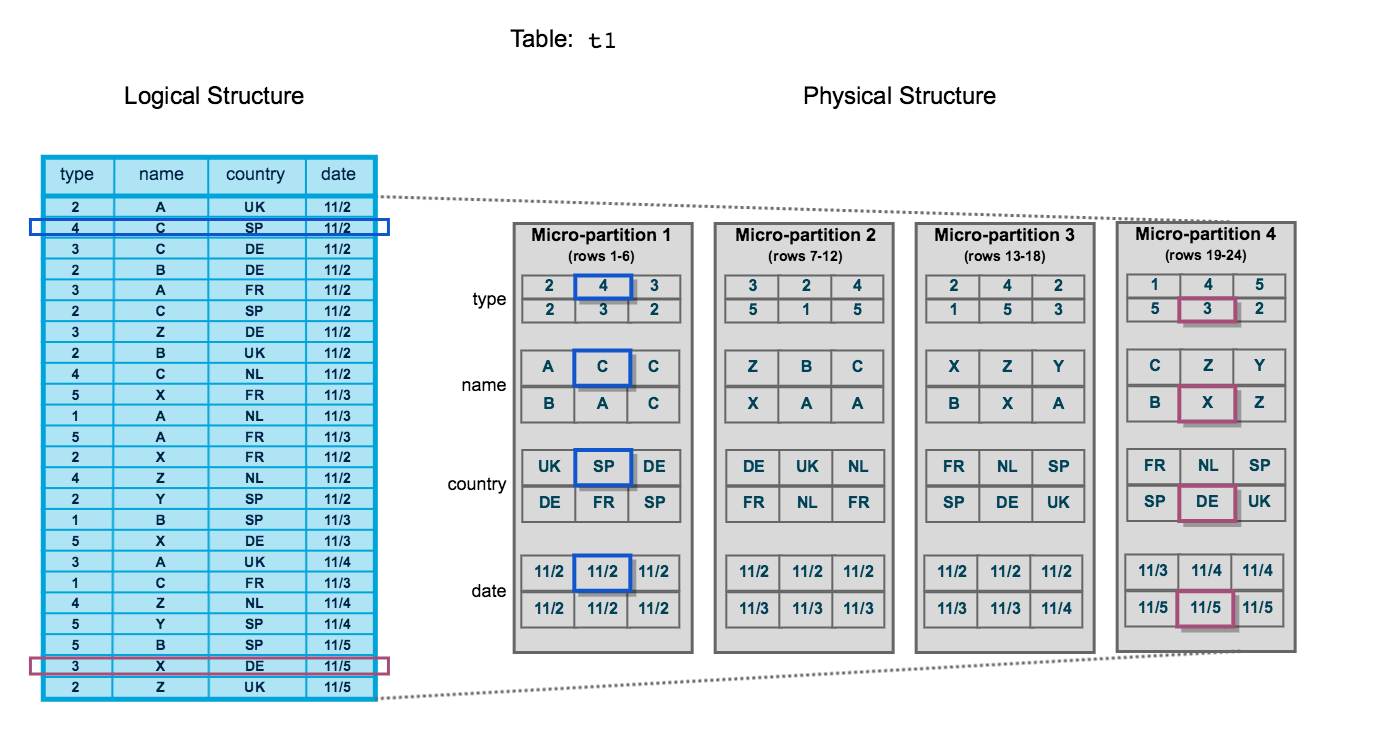
For time-series data, this level of pruning enables potentially sub-second response times for queries within ranges (i.e. “slices”) as fine-grained as one hour or even less.

What is Data Clustering?

Typically, data stored in tables is sorted/ordered along natural dimensions (e.g. date and/or geographic regions). This “clustering” is a key factor in queries because table data that is not sorted or is only partially sorted may impact query performance, particularly on very large tables.

In Snowflake, as data is inserted/loaded into a table, clustering metadata is collected and recorded for each micro-partition created during the process. Snowflake then leverages this clustering information to avoid unnecessary scanning of micro-partitions during querying, significantly accelerating the performance of queries that reference these columns.

The following diagram illustrates a Snowflake table, t1, with four columns sorted by date:



The table consists of 24 rows stored across 4 micro-partitions, with the rows divided equally between each micro-partition. Within each micro-partition, the data is sorted and stored by column, which enables Snowflake to perform the following actions for queries on the table:

1. First, prune micro-partitions that are not needed for the query.
2. Then, prune by column within the remaining micro-partitions.

Note that this diagram is intended only as a small-scale conceptual representation of the data clustering that Snowflake utilizes in micro-partitions. A typical Snowflake table may consist of thousands, even millions, of micro-partitions.

Clustering Information Maintained for Micro-partitions

Snowflake maintains clustering metadata for the micro-partitions in a table, including:

* The total number of micro-partitions that comprise the table.
* The number of micro-partitions containing values that overlap with each other (in a specified subset of table columns).
* The depth of the overlapping micro-partitions.

Clustering Depth

The clustering depth for a populated table measures the average depth (1 or greater) of the overlapping micro-partitions for specified columns in a table. The smaller the average depth, the better clustered the table is with regards to the specified columns.

Clustering depth can be used for a variety of purposes, including:

* Monitoring the clustering “health” of a large table, particularly over time as DML is performed on the table.
* Determining whether a large table would benefit from explicitly defining a [clustering key](https://docs.snowflake.net/manuals/user-guide/tables-clustering-keys.html).

A table with no micro-partitions (i.e. an unpopulated/empty table) has a clustering depth of 0.

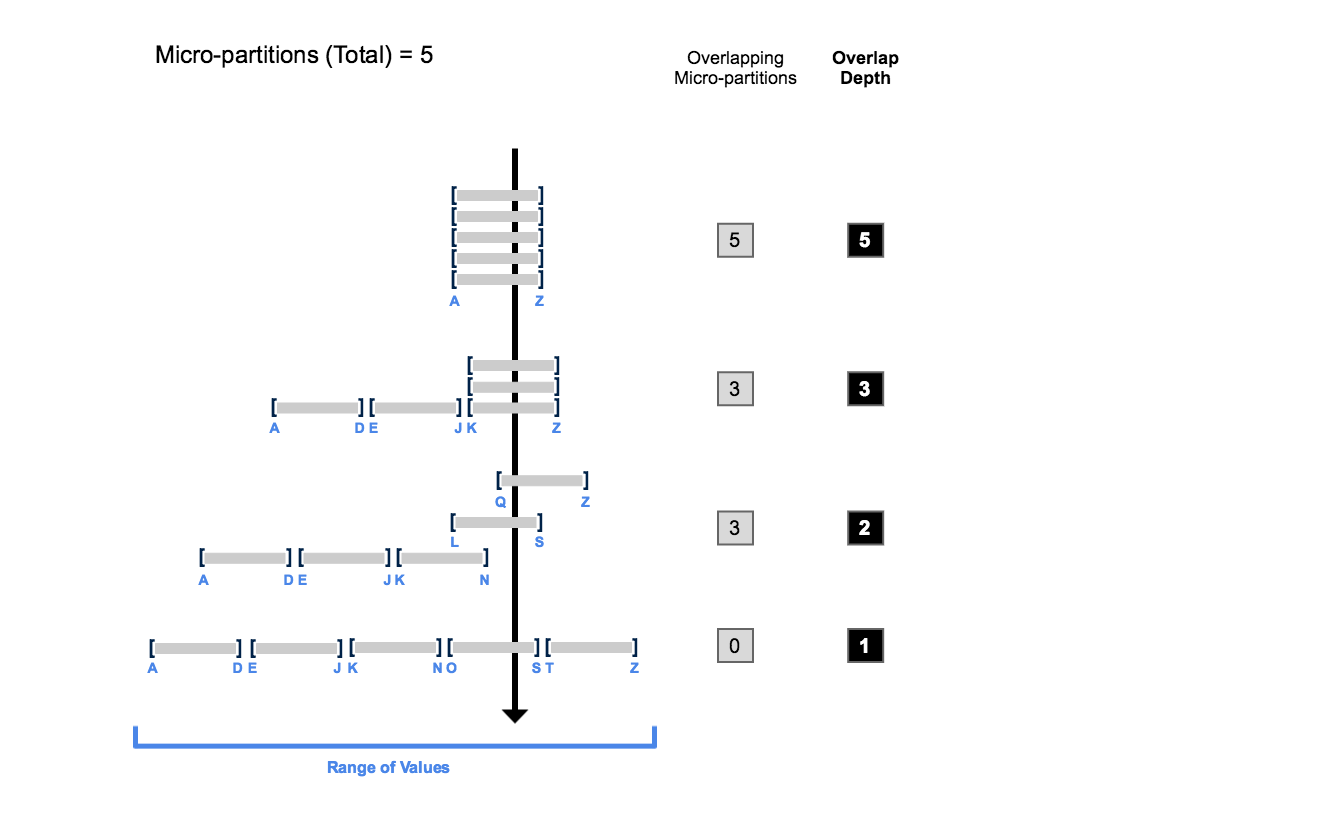
**Note**

The clustering depth for a table is ***not*** an absolute or precise measure of whether the table is well-clustered. Ultimately, query performance is the best indicator of how well-clustered a table is:

* If queries on a table are performing as needed or expected, the table is likely well-clustered.
* If query performance degrades over time, the table is likely no longer well-clustered and may benefit from clustering.

Clustering Depth Illustrated

The following diagram provides a conceptual example of a table consisting of five micro-partitions with values ranging from A to Z, and illustrates how overlap affects clustering depth:



As this diagram illustrates:

1. At the beginning, the range of values in all the micro-partitions overlap.
2. As the number of overlapping micro-partitions decreases, the overlap depth decreases.
3. When there is no overlap in the range of values across all micro-partitions, the micro-partitions are considered to be in a *constant state* (i.e. they cannot be improved by clustering).

The diagram is not intended to represent an actual table. In an actual table, with data contained in a large numbers of micro-partitions, reaching a constant state across all micro-partitions is neither likely nor required to improve query performance.

Monitoring Clustering Information for Tables

To view/monitor the clustering metadata for a table, Snowflake provides the following system functions:

* [SYSTEM$CLUSTERING\_DEPTH](https://docs.snowflake.net/manuals/sql-reference/functions/system_clustering_depth.html)
* [SYSTEM$CLUSTERING\_INFORMATION](https://docs.snowflake.net/manuals/sql-reference/functions/system_clustering_information.html) (including clustering depth)

For more details about how

SYSTEM$CLUSTERING\_DEPTH

Computes the average depth of the table according to the specified columns (or the clustering key defined for the table). The average depth of a populated table (i.e. a table containing data) is always 1 or more. The smaller the average depth, the better clustered the table is with regards to the specified columns.

For more information about micro-partitions and clustering keys, see [Understanding Snowflake Table Structures](https://docs.snowflake.net/manuals/user-guide/tables-micro-partitions.html).

**See also:**

[SYSTEM$CLUSTERING\_INFORMATION](https://docs.snowflake.net/manuals/sql-reference/functions/system_clustering_information.html)

Syntax

SYSTEM$CLUSTERING\_DEPTH( '*<table\_name>*' , '( *<col1>* [ , *<col2>* ... ] )' [ , '*<predicate>*' ] )

Arguments

***table\_name***

Table for which you want to calculate the clustering depth.

***col1* [ , *col2* ... ]**

Column(s) in the table used to calculate the clustering depth:

* For a table with no clustering key, this argument is required. If this argument is omitted, an error is returned.
* For a table with a clustering key, this argument is optional; if the argument is omitted, Snowflake uses the defined clustering key to calculate the depth.

**Note**

You can use this argument to calculate the depth for any columns in the table, regardless of the clustering key defined for the table.

***predicate***

Clause that filters the range of values in the columns on which to calculate the clustering depth. Note that *predicate* does not utilize a WHERE keyword at the beginning of the clause.

Usage Notes

* All arguments are strings (i.e. they must be enclosed in single quotes).
* If *predicate* contains a string, the string must be enclosed in single quotes, which then must be escaped using single quotes. For example:

SYSTEM$CLUSTERING\_DEPTH( ... , 'col1 = 100 and col2 = ''A''' )

Examples

Calculate the clustering depth for a table using the clustering key defined for the table:

select system$clustering\_depth('TPCH\_ORDERS');

+----------------------------------------+

| SYSTEM$CLUSTERING\_DEPTH('TPCH\_ORDERS') |

|----------------------------------------+

| 2.4865 |

+----------------------------------------+

Calculate the clustering depth for a table using two columns in the table:

select system$clustering\_depth('TPCH\_ORDERS', '(C2, C9)');

+----------------------------------------------------+

| SYSTEM$CLUSTERING\_DEPTH('TPCH\_ORDERS', '(C2, C9)') |

+----------------------------------------------------+

| 23.1351 |

+----------------------------------------------------+

Same as the previous example, but with a predicate on one of the columns:

select system$clustering\_depth('TPCH\_ORDERS', '(C2, C9)', 'C2 = 25');

+----------------------------------------------------+

| SYSTEM$CLUSTERING\_DEPTH('TPCH\_ORDERS', '(C2, C9)') |

+----------------------------------------------------+

| 11.2452 |

+----------------------------------------------------+

# SYSTEM$CLUSTERING\_INFORMATION

Returns clustering information, including average clustering depth, for a table based on one or more columns in the table.

**See also:**

[SYSTEM$CLUSTERING\_DEPTH](https://docs.snowflake.net/manuals/sql-reference/functions/system_clustering_depth.html)

Syntax

SYSTEM$CLUSTERING\_INFORMATION( '*<table\_name>*' , '( *<col1>* [ , *<col2>* ... ] )' )

Arguments

***table\_name***

Table for which you want to return clustering information.

***col1* [ , *col2* ... ]**

Column(s) in the table for which clustering information is returned:

* For a table with no clustering key, this argument is required. If this argument is omitted, an error is returned.
* For a table with a clustering key, this argument is optional; if the argument is omitted, Snowflake uses the defined clustering key to return clustering information.

**Note**

You can use this argument to return clustering information for any columns in the table, regardless of whether a clustering key is defined for the table.

In other words, you can use this to help you decide what clustering to use in the future.

Usage Notes

* All arguments are strings (i.e. they must be enclosed in single quotes).

Output

The function returns a JSON object containing the following name/value pairs:

**cluster\_by\_keys**

Columns in table used to return clustering information; can be any columns in the table.

**notes**

This column can contain suggestions to make clustering more efficient. For example, this field might contain a warning if the cardinality of the clustering column is extremely high.

This column can be empty.

For more information about how to cluster efficiently, see [Strategies for Selecting Clustering Keys](https://docs.snowflake.net/manuals/user-guide/tables-clustering-keys.html#label-clustering-keys-strategies).

**total\_partition\_count**

Total number of micro-partitions that comprise the table.

**total\_constant\_partition\_count**

Total number of micro-partitions for which the value of the specified columns have reached a constant state (i.e. the micro-partitions will not benefit significantly from reclustering). The number of constant micro-partitions in a table has an impact on pruning for queries. The higher the number, the more micro-partitions can be pruned from queries executed on the table, which has a corresponding impact on performance.

**average\_overlaps**

Average number of overlapping micro-partitions for each micro-partition in the table. A high number indicates the table is not well-clustered.

**average\_depth**

Average overlap depth of each micro-partition in the table. A high number indicates the table is not well-clustered.

This value is also returned by [SYSTEM$CLUSTERING\_DEPTH](https://docs.snowflake.net/manuals/sql-reference/functions/system_clustering_depth.html).

**partition\_depth\_histogram**

A histogram depicting the distribution of overlap depth for each micro-partition in the table. The histogram contains buckets with widths:

* 0 to 16 with increments of 1.
* For buckets larger than 16, increments of twice the width of the previous bucket (e.g. 32, 64, 128, …).

For more information about micro-partition overlap and depth and their impact on query pruning, see [Understanding Snowflake Table Structures](https://docs.snowflake.net/manuals/user-guide/tables-micro-partitions.html).

Examples

Return the clustering information for a table using two columns in the table:

select system$clustering\_information('test2', '(col1, col3)');

+--------------------------------------------------------------+

| SYSTEM$CLUSTERING\_INFORMATION('TEST2', '(COL1, COL3)') |

|--------------------------------------------------------------|

| { |

| "cluster\_by\_keys" : "(COL1, COL3)", |

| "total\_partition\_count" : 1156, |

| "total\_constant\_partition\_count" : 0, |

| "average\_overlaps" : 117.5484, |

| "average\_depth" : 64.0701, |

| "partition\_depth\_histogram" : { |

| "00000" : 0, |

| "00001" : 0, |

| "00002" : 3, |

| "00003" : 3, |

| "00004" : 4, |

| "00005" : 6, |

| "00006" : 3, |

| "00007" : 5, |

| "00008" : 10, |

| "00009" : 5, |

| "00010" : 7, |

| "00011" : 6, |

| "00012" : 8, |

| "00013" : 8, |

| "00014" : 9, |

| "00015" : 8, |

| "00016" : 6, |

| "00032" : 98, |

| "00064" : 269, |

| "00128" : 698 |

| } |

| } |

+--------------------------------------------------------------+

This example indicates that the test2 table is ***not*** well-clustered for the following reasons:

* Zero (0) constant micro-partitions out of 1156 total micro-partitions.
* High average of overlapping micro-partitions.
* High average of overlap depth across micro-partitions.
* Most of the micro-partitions are grouped at the lower-end of the histogram, with the majority of micro-partitions having an overlap depth between 64 and 128.

# Clustering Keys & Clustered Tables

In general, Snowflake produces well-clustered data in tables; however, over time, particularly as DML occurs on very large tables (as defined by the amount of data in the table, not the number of rows), the data in some table rows may no longer cluster optimally on desired dimensions.

To improve the clustering of the underlying table micro-partitions, you can always manually sort rows on key table columns and re-insert them into the table; however, performing these tasks could be cumbersome and expensive.

Instead, Snowflake supports automating these tasks by designating one or more table columns/expressions as a clustering key for the table. A table with a clustering key defined is considered to be clustered.

**Attention**

Clustering keys are ***not*** intended for all tables. The size of a table, as well as the query performance for the table, should dictate whether to define a clustering key for the table. In particular, to see performance improvements from a clustering key, a table has to be large enough to consist of a sufficiently large number of [micro-partitions](https://docs.snowflake.net/manuals/user-guide/tables-clustering-micropartitions.html), and the column(s) defined in the clustering key have to provide sufficient filtering to select a subset of these micro-partitions.

In general, tables in the multi-terabyte (TB) range will experience the most benefit from clustering, particularly if DML is performed regularly/continually on these tables.

Also, before explicitly choosing to cluster a table, Snowflake ***strongly*** recommends that you test a representative set of queries on the table to establish some performance baselines.

**Sections in this topic:**

* [What is a Clustering Key?](https://docs.snowflake.net/manuals/user-guide/tables-clustering-keys.html#what-is-a-clustering-key)
* [Benefits of Defining Clustering Keys (for Very Large Tables)](https://docs.snowflake.net/manuals/user-guide/tables-clustering-keys.html#benefits-of-defining-clustering-keys-for-very-large-tables)
* [Strategies for Selecting Clustering Keys](https://docs.snowflake.net/manuals/user-guide/tables-clustering-keys.html#strategies-for-selecting-clustering-keys)
* [Reclustering](https://docs.snowflake.net/manuals/user-guide/tables-clustering-keys.html#reclustering)
  + [Credit and Storage Impact of Reclustering](https://docs.snowflake.net/manuals/user-guide/tables-clustering-keys.html#credit-and-storage-impact-of-reclustering)
  + [Reclustering Example](https://docs.snowflake.net/manuals/user-guide/tables-clustering-keys.html#reclustering-example)
* [Defining Clustered Tables](https://docs.snowflake.net/manuals/user-guide/tables-clustering-keys.html#defining-clustered-tables)
  + [Calculating the Clustering Information for a Table](https://docs.snowflake.net/manuals/user-guide/tables-clustering-keys.html#calculating-the-clustering-information-for-a-table)
  + [Defining a Clustering Key for a Table](https://docs.snowflake.net/manuals/user-guide/tables-clustering-keys.html#defining-a-clustering-key-for-a-table)
    - [Important Usage Notes](https://docs.snowflake.net/manuals/user-guide/tables-clustering-keys.html#important-usage-notes)
  + [Changing the Clustering Key for a Table](https://docs.snowflake.net/manuals/user-guide/tables-clustering-keys.html#changing-the-clustering-key-for-a-table)
    - [Important Usage Notes](https://docs.snowflake.net/manuals/user-guide/tables-clustering-keys.html#id1)
  + [Dropping the Clustering Keys for a Table](https://docs.snowflake.net/manuals/user-guide/tables-clustering-keys.html#dropping-the-clustering-keys-for-a-table)

## What is a Clustering Key?

A clustering key is a subset of columns in a table (or expressions on a table) that are explicitly designated to co-locate the data in the table in the same [micro-partitions](https://docs.snowflake.net/manuals/user-guide/tables-clustering-micropartitions.html). This is useful for very large tables where the ordering was not ideal (at the time the data was inserted/loaded) or extensive DML has caused the table’s natural clustering to degrade.

Some general indicators that can help determine whether to define a clustering key for a table include:

* Queries on the table are running slower than expected or have noticeably degraded over time.
* The [clustering depth](https://docs.snowflake.net/manuals/user-guide/tables-clustering-micropartitions.html#label-clustering-depth) for the table is large.

A clustering key can be defined at table creation (using the [CREATE TABLE](https://docs.snowflake.net/manuals/sql-reference/sql/create-table.html) command) or afterward (using the [ALTER TABLE](https://docs.snowflake.net/manuals/sql-reference/sql/alter-table.html) command). The clustering key for a table can also be altered or dropped at any time.

## Benefits of Defining Clustering Keys (for Very Large Tables)

Using a clustering key to co-locate similar rows in the same micro-partitions enables several benefits for very large tables, including:

* Improved scan efficiency in queries by skipping data that does not match filtering predicates.
* Better column compression than in tables with no clustering. This is especially true when other columns are strongly correlated with the columns that comprise the clustering key.
* After a key has been defined on a table, no additional administration is required, unless you chose to drop or modify the key. All future maintenance on the rows in the table (to ensure optimal clustering) is performed automatically by Snowflake.

Although clustering can substantially improve the performance and reduce the cost of some queries, the compute resources used to perform clustering consume credits. As such, you should cluster only when queries will benefit substantially from the clustering.

Typically, queries benefit from clustering when the queries filter or sort on the clustering key for the table. Sorting is commonly done for ORDER BY operations, for GROUP BY operations, and for some joins. For example, the following join would likely cause Snowflake to perform a sort operation:

select ...

from my\_table inner join my\_materialized\_view

on my\_materialized\_view.col1 = my\_table.col1

...

In this pseudo-example, Snowflake is likely to sort the values in either my\_materialized\_view.col1 or my\_table.col1. For example, if the values in my\_table.col1 are sorted, then as the materialized view is being scanned, Snowflake can quickly find the corresponding row in my\_table.

The more frequently a table is queried, the more benefit you’ll get from clustering. However, the more frequently a table changes, the more expensive it will be to keep it clustered. Therefore, clustering is generally most cost-effective for tables that are queried frequently and do not change frequently.

**Note**

After you define a clustering key for a table, the rows are not necessarily updated immediately. Snowflake only performs automated maintenance if the table will benefit from the operation. For more details, see [Reclustering](https://docs.snowflake.net/manuals/user-guide/tables-clustering-keys.html" \l "reclustering) (in this topic) and [Automatic Clustering](https://docs.snowflake.net/manuals/user-guide/tables-auto-reclustering.html).

## Strategies for Selecting Clustering Keys

Selecting the right columns/expressions for a clustering key can dramatically impact query performance. Analysis of your workload will usually yield good clustering key candidates.

Snowflake recommends prioritizing keys in the order below:

1. Cluster columns that are most actively used in selective filters. For many fact tables involved in date-based queries (for example “WHERE invoice\_date > x AND invoice date <= y”), choosing the date column is a good idea. For event tables, event type might be a good choice, if there are a large number of different event types. (If your table has only a small number of different event types, then see the comments on cardinality below before choosing an event column as a clustering key.)
2. If there is room for additional cluster keys, then consider columns frequently used in join predicates, for example “FROM table1 JOIN table2 ON table2.column\_A = table1.column\_B”.

If you typically filter queries by two dimensions (e.g. application\_id and user\_status columns), then clustering on both columns can improve performance.

The number of distinct values (i.e. cardinality) in a column/expression is a critical aspect of selecting it as a clustering key. It is important to choose a clustering key that has:

* A large enough number of distinct values to enable effective pruning on the table.
* A small enough number of distinct values to allow Snowflake to effectively group rows in the same micro-partitions.

A column with very low cardinality (e.g. a column that indicates only whether a person is male or female) might yield only minimal pruning. At the other extreme, a column with very high cardinality (e.g. a column containing UUID or nanosecond timestamp values) is also typically ***not*** a good candidate to use as a clustering key directly.

**Tip**

In general, if a column (or expression) has higher cardinality, then maintaining clustering on that column is more expensive.

The cost of clustering on a unique key might be more than the benefit of clustering on that key, especially if point lookups are not the primary use case for that table.

If you want to use a column with very high cardinality as a clustering key, Snowflake recommends defining the key as an expression on the column, rather than on the column directly, to reduce the number of distinct values. The expression should preserve the original ordering of the column so that the minimum and maximum values in each partition still enable pruning.

For example, if a fact table has a TIMESTAMP column c\_timestamp containing many discrete values (many more than the number of micro-partitions in the table), then a clustering key could be defined on the column by casting the values to dates instead of timestamps (e.g. to\_date(c\_timestamp)). This would reduce the cardinality to the total number of days, which typically produces much better pruning results.

As another example, you can truncate a number to fewer significant digits by using the TRUNC functions and a negative value for the scale, e.g., TRUNC(123456789, -5).

**Tip**

If you are defining a multi-column clustering key for a table, the order in which the columns are specified in the CLUSTER BY clause is important. As a general rule, Snowflake recommends ordering the columns from ***lowest*** cardinality to ***highest*** cardinality. Putting a higher cardinality column before a lower cardinality column will generally reduce the effectiveness of clustering on the latter column.

In some cases, clustering on columns used in GROUP BY or ORDER BY clauses can be helpful. However, clustering on these columns is usually less helpful than clustering on columns that are heavily used in filter or JOIN operations. If you have some columns that are heavily used in filter/join operations and different columns that are used in ORDER BY or GROUP BY operations, then favor the columns used in the filter and join operations.

## Reclustering

As DML operations (INSERT, UPDATE, DELETE, MERGE, COPY) are performed on a clustered table, the data in the table may become less clustered. Periodic/regular reclustering of the table is required to maintain optimal clustering.

During reclustering, Snowflake uses the clustering key for a clustered table to reorganize the column data, so that related records are relocated to the same micro-partition. This DML operation deletes the affected records and re-inserts them, grouped according to the clustering key.

**Note**

Reclustering in Snowflake is automatic; no maintenance is needed. For more details, see [Automatic Clustering](https://docs.snowflake.net/manuals/user-guide/tables-auto-reclustering.html).

However, for certain accounts, manual reclustering has been deprecated, but is still allowed. For more details see [Manual Reclustering](https://docs.snowflake.net/manuals/user-guide/tables-clustering-manual.html).

### Credit and Storage Impact of Reclustering

Similar to all DML operations in Snowflake, reclustering consumes credits. The number of credits consumed depends on the size of the table and the amount of data that needs to be reclustered.

Reclustering also results in storage costs. Each time data is reclustered, the rows are physically grouped based on the clustering key for the table, which results in Snowflake generating ***new*** micro-partitions for the table. Adding even a small number of rows to a table can cause all micro-partitions that contain those values to be recreated.

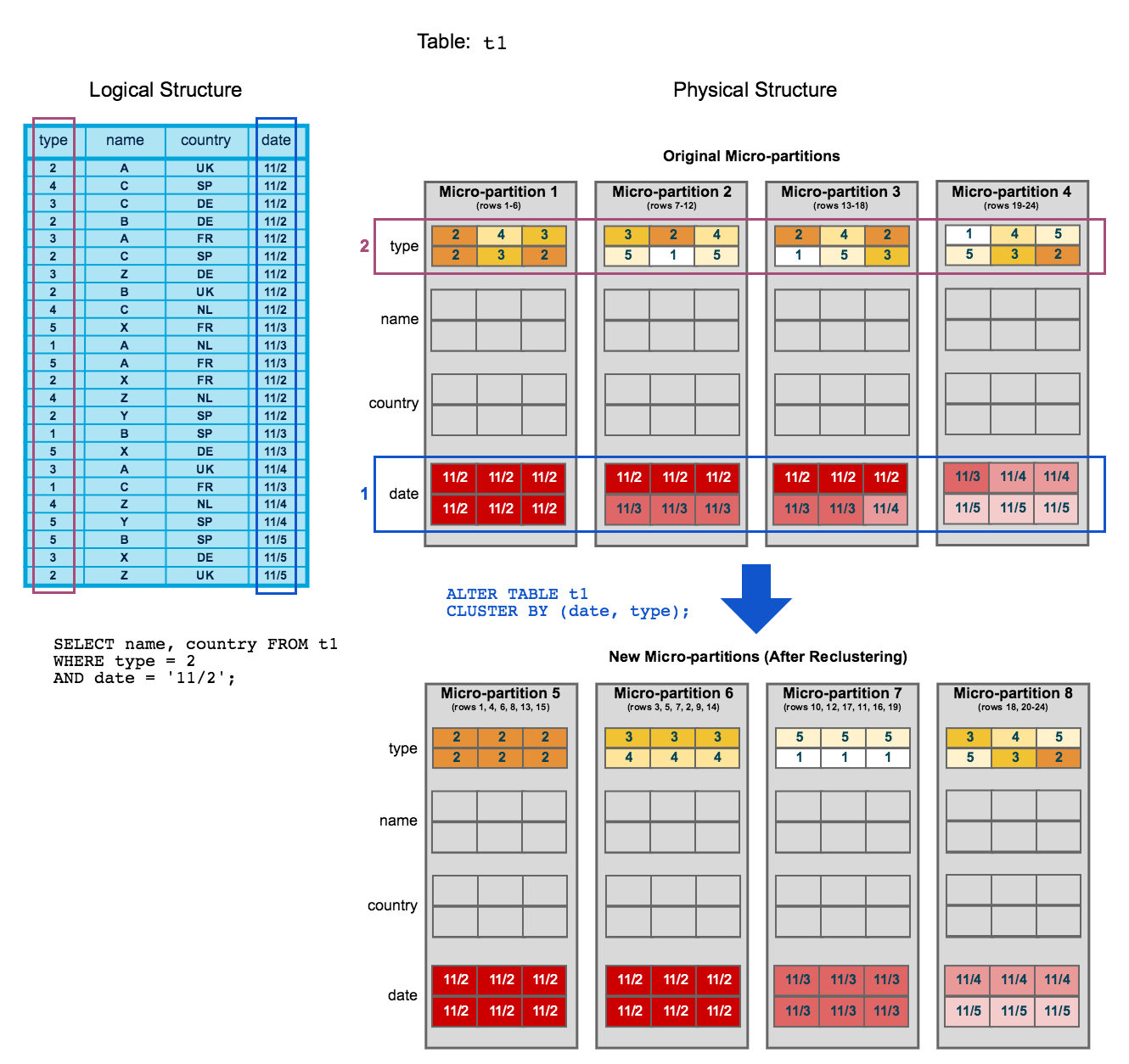
This process can create significant data turnover because the original micro-partitions are marked as deleted, but retained in the system to enable Time Travel and Fail-safe. The original micro-partitions are purged only after both the Time Travel retention period and the subsequent Fail-safe period have passed (i.e. minimum of 8 days and up to 97 days for extended Time Travel, if you are using Snowflake Enterprise Edition (or higher). This typically results in increased storage costs. For more information, see [Snowflake Time Travel & Fail-safe](https://docs.snowflake.net/manuals/user-guide/data-availability.html).

**Important**

Before defining a clustering key for a table, you should consider the associated credit and storage costs.

### Reclustering Example

Building on the [clustering diagram](https://docs.snowflake.net/manuals/user-guide/tables-clustering-micropartitions.html#label-data-clustering) from the previous topic, this diagram illustrates how reclustering a table can help reduce scanning of micro-partitions to improve query performance:



* To start, table t1 is naturally clustered by date across micro-partitions 1-4.
* The query (in the diagram) requires scanning micro-partitions 1, 2, and 3.
* date and id are defined as the clustering key. When the table is reclustered, new micro-partitions (5-8) are created.
* After reclustering, the same query only scans micro-partitions 5 and 6.

In addition, after reclustering:

* Micro-partition 5 has reached a constant state (i.e. it cannot be improved by reclustering) and is therefore excluded when computing depth and overlap for future maintenance. In a well-clustered large table, most micro-partitions will fall into this category.
* The original micro-partitions (1-4) are marked as deleted, but are ***not*** purged from the system; they are retained for [Time Travel and Fail-safe](https://docs.snowflake.net/manuals/user-guide/data-availability.html).

**Note**

This example illustrates the impact of reclustering on an extremely small scale. Extrapolated to a very large table (i.e. consisting of millions of micro-partitions or more), reclustering can have a significant impact on scanning and, therefore, query performance.

## Defining Clustered Tables

### Calculating the Clustering Information for a Table

Use the system function, [SYSTEM$CLUSTERING\_INFORMATION](https://docs.snowflake.net/manuals/sql-reference/functions/system_clustering_information.html), to calculate clustering details, including clustering depth, for a given table. This function can be run on any columns on any table, regardless of whether the table has an explicit clustering key:

* If a table has an explicit clustering key, the function doesn’t require any input arguments other than the name of the table.
* If a table doesn’t have an explicit clustering key (or a table has a clustering key, but you want to calculate the ratio on other columns in the table), the function takes the desired column(s) as an additional input argument.

### Defining a Clustering Key for a Table

A clustering key can be defined when a table is created by appending a CLUSTER BY clause to [CREATE TABLE](https://docs.snowflake.net/manuals/sql-reference/sql/create-table.html):

CREATE TABLE *<name>* ... CLUSTER BY ( *<expr1>* [ , *<expr2>* ... ] )

Where each clustering key consists of one or more table columns/expressions, which can be of any data type, ***except*** VARIANT, OBJECT, or ARRAY. A clustering key can contain any of the following:

* Base columns.
* Expressions on base columns.
* Expressions on paths in VARIANT columns.

For example:

*-- cluster by base columns*

create or replace table t1 (c1 date, c2 string, c3 number) cluster by (c1, c2);

show tables like 't1';

+-------------------------------+------+---------------+-------------+-------+---------+----------------+------+-------+----------+----------------+----------------------+

| created\_on | name | database\_name | schema\_name | kind | comment | cluster\_by | rows | bytes | owner | retention\_time | automatic\_clustering |

|-------------------------------+------+---------------+-------------+-------+---------+----------------+------+-------+----------+----------------+----------------------|

| 2019-06-20 12:06:07.517 -0700 | T1 | TESTDB | PUBLIC | TABLE | | LINEAR(C1, C2) | 0 | 0 | SYSADMIN | 1 | ON |

+-------------------------------+------+---------------+-------------+-------+---------+----------------+------+-------+----------+----------------+----------------------+

*-- cluster by expressions*

create or replace table t2 (c1 timestamp, c2 string, c3 number) cluster by (to\_date(c1), substring(c2, 0, 10));

show tables like 't2';

+-------------------------------+------+---------------+-------------+-------+---------+------------------------------------------------+------+-------+----------+----------------+----------------------+

| created\_on | name | database\_name | schema\_name | kind | comment | cluster\_by | rows | bytes | owner | retention\_time | automatic\_clustering |

|-------------------------------+------+---------------+-------------+-------+---------+------------------------------------------------+------+-------+----------+----------------+----------------------|

| 2019-06-20 12:07:51.307 -0700 | T2 | TESTDB | PUBLIC | TABLE | | LINEAR(CAST(C1 AS DATE), SUBSTRING(C2, 0, 10)) | 0 | 0 | SYSADMIN | 1 | ON |

+-------------------------------+------+---------------+-------------+-------+---------+------------------------------------------------+------+-------+----------+----------------+----------------------+

*-- cluster by paths in variant columns*

create or replace table t3 (t timestamp, v variant) cluster by (v:"Data":id::number);

show tables like 'T3';

+-------------------------------+------+---------------+-------------+-------+---------+-------------------------------------------+------+-------+----------+----------------+----------------------+

| created\_on | name | database\_name | schema\_name | kind | comment | cluster\_by | rows | bytes | owner | retention\_time | automatic\_clustering |

|-------------------------------+------+---------------+-------------+-------+---------+-------------------------------------------+------+-------+----------+----------------+----------------------|

| 2019-06-20 16:30:11.330 -0700 | T3 | TESTDB | PUBLIC | TABLE | | LINEAR(TO\_NUMBER(GET\_PATH(V, 'Data.id'))) | 0 | 0 | SYSADMIN | 1 | ON |

+-------------------------------+------+---------------+-------------+-------+---------+-------------------------------------------+------+-------+----------+----------------+----------------------+

#### **Important Usage Notes**

* If you define two or more columns/expressions as the clustering key for a table, the order has an impact on how the data is clustered in micro-partitions.

For more details, see [Strategies for Selecting Clustering Keys](https://docs.snowflake.net/manuals/user-guide/tables-clustering-keys.html#strategies-for-selecting-clustering-keys) (in this topic).

* An existing clustering key is copied when a table is created using CREATE TABLE … CLONE.
* An existing Clustering key is ***not*** propagated when a table is created using CREATE TABLE … LIKE.
* An existing clustering key is ***not*** supported when a table is created using CREATE TABLE … AS SELECT; however, you can define a clustering key after the table is created.
* Defining a clustering key directly on top of VARIANT columns are not supported; however, you can specify a VARIANT column in a clustering key if you provide an expression consisting of the path and the target type.

### Changing the Clustering Key for a Table

At any time, you can add a clustering key to an existing table or change the existing clustering key for a table using [ALTER TABLE](https://docs.snowflake.net/manuals/sql-reference/sql/alter-table.html):

ALTER TABLE *<name>* CLUSTER BY ( *<expr1>* [ , *<expr2>* ... ] )

For example:

*-- cluster by base columns*

alter table t1 cluster by (c1, c3);

show tables like 't1';

+-------------------------------+------+---------------+-------------+-------+---------+----------------+------+-------+----------+----------------+----------------------+

| created\_on | name | database\_name | schema\_name | kind | comment | cluster\_by | rows | bytes | owner | retention\_time | automatic\_clustering |

|-------------------------------+------+---------------+-------------+-------+---------+----------------+------+-------+----------+----------------+----------------------|

| 2019-06-20 12:06:07.517 -0700 | T1 | TESTDB | PUBLIC | TABLE | | LINEAR(C1, C3) | 0 | 0 | SYSADMIN | 1 | ON |

+-------------------------------+------+---------------+-------------+-------+---------+----------------+------+-------+----------+----------------+----------------------+

*-- cluster by expressions*

alter table t2 cluster by (substring(c2, 5, 15), to\_date(c1));

show tables like 't2';

+-------------------------------+------+---------------+-------------+-------+---------+------------------------------------------------+------+-------+----------+----------------+----------------------+

| created\_on | name | database\_name | schema\_name | kind | comment | cluster\_by | rows | bytes | owner | retention\_time | automatic\_clustering |

|-------------------------------+------+---------------+-------------+-------+---------+------------------------------------------------+------+-------+----------+----------------+----------------------|

| 2019-06-20 12:07:51.307 -0700 | T2 | TESTDB | PUBLIC | TABLE | | LINEAR(SUBSTRING(C2, 5, 15), CAST(C1 AS DATE)) | 0 | 0 | SYSADMIN | 1 | ON |

+-------------------------------+------+---------------+-------------+-------+---------+------------------------------------------------+------+-------+----------+----------------+----------------------+

*-- cluster by paths in variant columns*

alter table t3 cluster by (v:"Data":name::string, v:"Data":id::number);

show tables like 'T3';

+-------------------------------+------+---------------+-------------+-------+---------+------------------------------------------------------------------------------+------+-------+----------+----------------+----------------------+

| created\_on | name | database\_name | schema\_name | kind | comment | cluster\_by | rows | bytes | owner | retention\_time | automatic\_clustering |

|-------------------------------+------+---------------+-------------+-------+---------+------------------------------------------------------------------------------+------+-------+----------+----------------+----------------------|

| 2019-06-20 16:30:11.330 -0700 | T3 | TESTDB | PUBLIC | TABLE | | LINEAR(TO\_CHAR(GET\_PATH(V, 'Data.name')), TO\_NUMBER(GET\_PATH(V, 'Data.id'))) | 0 | 0 | SYSADMIN | 1 | ON |

+-------------------------------+------+---------------+-------------+-------+---------+------------------------------------------------------------------------------+------+-------+----------+----------------+----------------------+

#### **Important Usage Notes**

* When adding a clustering key to a table already populated with data, not all expressions are allowed to be specified in the key. You can check whether a specific function is supported using [SHOW FUNCTIONS](https://docs.snowflake.net/manuals/sql-reference/sql/show-functions.html):

show functions like '*function\_name*';

The output includes a column, valid\_for\_clustering, at the end of the output. This column displays whether the function can be used in a clustering key for a populated table.

* Changing the clustering key for a table does not affect existing records in the table until the table has been reclustered by Snowflake.

### Dropping the Clustering Keys for a Table

At any time, you can drop the clustering key for a table using [ALTER TABLE](https://docs.snowflake.net/manuals/sql-reference/sql/alter-table.html):

ALTER TABLE *<name>* DROP CLUSTERING KEY

For example:

alter table t1 drop clustering key;

show tables like 't1';

+-------------------------------+------+---------------+-------------+-------+---------+------------+------+-------+----------+----------------+----------------------+

| created\_on | name | database\_name | schema\_name | kind | comment | cluster\_by | rows | bytes | owner | retention\_time | automatic\_clustering |

|-------------------------------+------+---------------+-------------+-------+---------+------------+------+-------+----------+----------------+----------------------|

| 2019-06-20 12:06:07.517 -0700 | T1 | TESTDB | PUBLIC | TABLE | | | 0 | 0 | SYSADMIN | 1 | OFF |

+-------------------------------+------+---------------+-------------+-------+---------+------------+------+-

