# Data Modeling with Snowflake: A Practical Guide to Accelerating Snowflake Development Using Universal Data Modeling Techniques

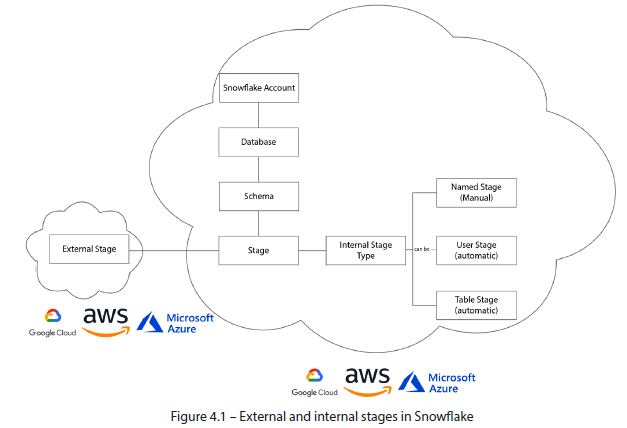
## Part 1: Core Concepts in Data Modeling and Snowflake Architecture

### Chapter 4: Mastering Snowflake Objects

* The previous chapter introduced the innovative **architecture** that powers the **Data Cloud** + unlocks possibilities that have never been possible in other databases
* Now we explore the **database objects** that Snowflake customers will use in their modeling journey
* Objects such as **tables** + **views** are familiar to most people who have previously worked w/ SQL
* Still, even here, Snowflake’s unique features unlock potential efficiencies in designing physical models, so users should be well acquainted w/ them
* Due to its **variable-spend pricing**, Snowflake data modeling **requires users to be well acquainted w/ the cost/performance trade-offs of its supported objects + their parameters**
* Exclusive features such as **streams** and **external tables** may still be unfamiliar
* But making good design decisions demands that users have a firm grasp + a thorough understanding of Snowflake’s complete modeling toolset
* **Main Topics**
* **Stages** and **file format** objects used for loading data into Snowflake
* The various **physical table types** used to store data
* **Metadata tables** for reviewing staged files
* **Views** and associated cost and performance benefits
* **Materialized views** and when to use them
* **Streams** for **data change tracking**
* **Tasks** for automating continuous ETL pipelines

#### Stages

* Before data can make its way into tables, it **must be loaded into Snowflake using** **Stages**, which are **logical objects that abstract cloud filesystems so they can be used in a standard manner to load data into Snowflake**
* There are **2 types of stages** that can be defined: **external** and **internal**
* **External stages** can be **created on top of a cloud location (for the supported cloud storage services) *outside* Snowflake**
* **Used to load data from *external* source systems**
* Supported cloud storage services for external stages are as: Amazon S3 buckets, Google Cloud Storage (GCS) buckets, + Microsoft Azure containers
* **Internal stages** are **created *within* a Snowflake account + therefore use the storage type of the hosting provider**
* Used to **stage files that originate from *w/in* the Snowflake account + can NOT be used to load external data**
* The following figure shows the layout of external stages in relation to internal stage types.



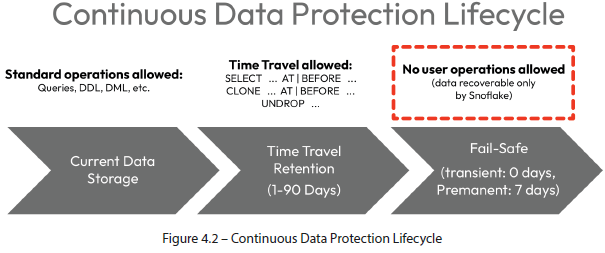
* **Unlike the data stored in tables, which is micro-partitioned + uses an internal columnar format, stages can store ANY type of file**
* Although only certain structured + semi-structured file types are supported for loading data into Snowflake, the ***visibility* of unstructured files** in a stage is an important consideration when paired with external and directory tables (discussed later)
* So, **stages can store any variety of files + file types *BUT* a file format *must* be specified when accessing data from a stage for subsequent loading into a Snowflake table**

#### File Formats

* **File formats** can be thought of as **templates describing the file types for reading or writing data to a file**
* **It** **defines properties** such as delimiter type (e.g., comma, tab, or other), date/time formats, and encoding (ex: Hex Base64, UTF-8, etc.)
* **Snowflake provides several structured + *semi*-structured file formats**, such as CSV, JSON, + Parquet, w/ commonly used defaults
* However, **users are free to create their own named file formats when loading or unloading files of a specified type**
* File format **properties may also be specified at runtime when using the COPY** **INTO** **command**
* A description of supported file types + file formats for data loading is available in the documentation: <https://docs.snowflake.com/en/user-guide/data-load-prepare.html>
* For a complete overview of data loading of all kinds, including streaming, see the guide on the Snowflake website: <https://docs.snowflake.com/en/user-guide-data-load.html>
* Having seen how data is loaded from stages into Snowflake using file formats, let’s get to know the **tables** that will be used to store it

#### Tables

* **Data in Snowflake is stored in tables**, which, as discussed, are **one of the fundamental components of data modeling**
* However, before exploring them in a modeling context, we should understand the various table types that exist in Snowflake + their costs
* The previous chapter described **Snowflake’s** **Time Travel** feature which **allows restoring dropped objects or querying data at a prior point in time**
* However, **Time Travel comes w/ associated storage costs, + the number of available Time Travel days (the retention period)** ***depends on the table type***, as we’ll shortly review in detail
* Snowflake also offers a ***managed* type of Time Travel**, known as **Fail-safe**
* All **permanent tables** have a **Fail-safe period of 7 days**
* ***Unlike* Time Travel, which the user can access,** **Fail-safe is managed by + accessible only to Snowflake to protect user data from disasters such as system failures + data breaches**
* **To recover data stored in Fail-safe, users need to contact Snowflake directly**
* For tables that offer **Time Travel *and* Fail-safe, the 7-day Fail-safe period begins immediately after the Time Travel period ends**
* The figure below illustrates the **types + duration of data protection that Snowflake tables offer:**



* **Different use cases call for different backup strategies**
* Transactional data may require days of history, while a staging table may be needed only temporarily
* **Snowflake offers several physical table types w/varying retention (+ associated cost) options**:
* **Permanent table**
* **Transient table**
* **Temporary table**
* **Hybrid Unistore tables**
* **Snowflake also provides several metadata table constructs for stage contents**
* These are **semantic objects that exist on top of stages to allow users to query file contents and metadata (such as filenames) just like they would a regular table**
* These objects are **read-only and are offered in the following formats**:
* **External tables**
* **Directory tables**

##### a) Physical Tables

* **In Snowflake, data is stored in physical tables logically grouped by schemas**
* **Tables consist of columns w/ names, data types, + optional constraints + properties** (ex: nullability, default value, + primary key (PK))
* The DDL command for creating a table is **CREATE TABLE**, which is **used in the relational modeling scenario to instantiate an empty table w/ a given structure**
* **Other variants of CREATE TABLE are used in the transformational context to create, pattern, or clone tables from *existing* objects w/ corresponding data, where applicable**
* These **operations are referred to as *transformational* b/cthey require a relational model/schema to exist *beforehand* (ex: creating a table by cloning an existing table)**
* The **supported variants** are as follows:
* **CREATE TABLE AS SELECT**: Commonly referred to as **CTAS**, **creates a table containing the data returned from a query (+ using the *implied* data types)**
* **CREATE TABLE USING TEMPLATE**: Creates an **empty table using column definitions derived from semi-structured files** (currently supported for **Parquet**, **Avro**, + **ORC** formats)
* **CREATE TABLE LIKE:** Creates an **empty table using the column definitions of an existing table**, + **column properties, defaults, + constraints are also copied**
* **CREATE TABLE CLONE:** Does the **same as LIKE *but*** **uses zero-copy cloning to *also* include all the data from the source table (*w/out actually copying it*)**
* This **CLONE variant can be used w/ Time Travel to clone the data at a specific point in the past**
* Again, **cloned tables are virtual copies of a source + do NOT consume storage *until the structure or contents are modified in the clone OR source objects***

###### 1) Permanent Tables

* Snowflake’s ***default* table type is a permanent table** (a slight misnomer since **all Snowflake tables except for temporary tables persist until explicitly dropped**)
* Permanent tables **come w/ built-in backup features**
* Includes a Fail-safe period of 7 days that *cannot* be modified or disabled, + a default of 1 day of Time Travel
* Time Travel period for permanent tables can be disabled or extended on Snowflake’s Enterprise plan to upward of 90 days
* Their **extensive recoverability makes permanent tables an ideal candidate for storing business-critical information that would be costly or impossible to regenerate in the event of a catastrophe**
* We can use the following command to create a permanent table: CREATE TABLE <table-name>
* However, ***not all tables require backups or extended Time Travel periods***
* In such cases, **transient tables should be considered instead**

###### 2) Transient Tables

* Unlike permanent tables, **transient tables have NO Fail-safe period but *DO* come w/ 1 day of Time Travel by default**
* **Can turn off Time Travel by setting it to 0, but** **cannot extend it beyond 1**, even on Enterprise plans
* Transient tables are a **good choice when backup + recovery are *not* a high priority, as in the case of staging tables**
* **The same is true for DEV and TEST environments, where transient defaults can be set at the schema *and* database level** (described later)
* The command for creating a transient table is: CREATE TRANSIENT TABLE <table-name>
* While **transient tables are ideal for staging data + DEV environments**, Snowflake also offers a **temporary table type that persists only for the duration of a user session**.

###### 3) Temporary Tables

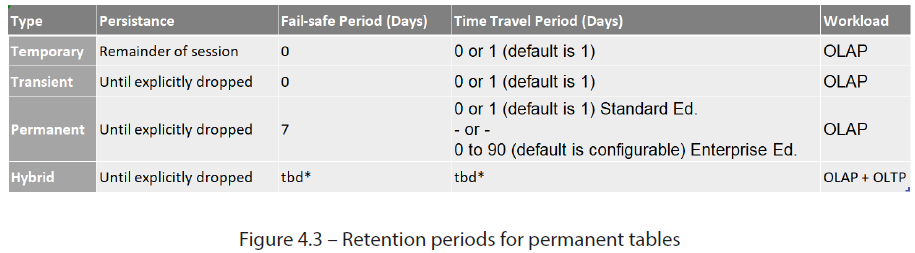
* The **data retention properties of temporary tables are identical to those of transient tables, aside from *one* *critical difference*: temporary tables are dropped at the end of the user session**
* A **session** can be thought of as **an active login**
* Although **multiple sessions (such as when using multiple worksheets) can run in parallel, they all terminate along w/ the connection or login**
* Due to their impermanent nature, **temporary tables are meant to hold intermediate results for processes *during the same session***
* Can also be used for **throw-away tests in DEV environments, saving the user the labor of dropping them explicitly**
* The command for creating a temporary table is: CREATE TEMPORARY TABLE <table-name>
* Note on temporary table naming:
* Although temporary tables *must* be assigned to a schema like all database objects, their **session-based persistence exempts them from uniqueness collisions w/ other table types**
* A **temporary table can have the same name as a non-temporary table + will take precedence over it for the remainder of the session (thereby hiding the non-temporary table)**

###### 4) Hybrid Unistore Tables

* **Hybrid Unistore table** is another **physical table** offered by Snowflake whose **properties have less to do w/ retention times than being a total game changer for the types of workloads that Snowflake is able to support**
* Announced at Snowflake Summit 2022 + currently offered as private previews for access
* Likely to be released in 2024, included here in anticipation of game-changing possibilities they’re poised to unlock
* **Hybrid Tables in Snowflake = 2 tables transparently managed under a single semantic object**
* **Consists of a standard (permanent) OLAP table + an OLTP table that the Snowflake services layer manages synchronously + transparently**
* The **combo of OLAP + OLTP** is known as **hybrid transactional and analytical processing (HTAP)** , and it **offers the best of both worlds**
* The **OLAP table will use the standard columnar format of the permanent tables described previously + provide all the analytical performance features that Snowflake is known for**
* The **OLTP table offers exclusive features that are NOT found in OLAP tables** (covered in next chapter), including the following:
* **Constraint enforcement**
* **Support for indexes**
* **Row-level locking for changes + updates**
* Snowflake ***synchronously* updates the OLAP + OLTP tables when data is loaded into a Hybrid Table**
* The **query optimizer then redirects DML or DQL queries to the table *best suited to the required operation***
* Although complete documentation for Hybrid Tables is not publicly available, the CREATE command has been shared + uses existing syntax for declaring constraints:
* CREATE HYBRID TABLE <name>

###### Table Types Summary

* When it comes to flexibility in selecting the right backup + workload for a table, Snowflake gives its users plenty to choose from when it comes to **physical tables + retention periods**



* We have gone into detail on Time Travel + recovery, but it is worth mentioning that **another type of history can also be enabled for physical tables: change history:**
* Unlike **Time Travel (allows users to access a historical snapshot of a table), change history** **records *individual* table changes + enables change data capture (CDC)**, a functionality covered in greater detail later

##### b) Stage Metadata Tables

* **Physical tables are what Snowflake users most often interact w/ for storing + manipulating information**, but other table types can be defined
* **External tables + directory tables (metadata tables over stage objects)** can be created + modeled relationally
* **External + directory tables** exist in Snowflake to **allow users to access data in staged files *as though selecting from regular tables***
* **External tables allow PKs + FKs and can be modeled in a standard relational context**
* **Directory tables are similar to external tables but do NOT allow the creation of columns or constraints, + are included for completeness**

###### i) External Tables

* **External tables are metadata objects created on top of stages that allow users to query the data in *externally* staged files *as if it were inside the database***
* **The same file formats supported by a stage are supported for accessing the data through external tables**
* The following **columns are included for all external tables**:
* **VALUE**: A VARIANT-type column **representing a single row in the external file**
* **METADATA$FILENAME**: **Identifies the name of each staged data file in the external table, including its path in the stage**
* **METADATA$FILE\_ROW\_NUMBER**: Shows the **row number for each record in a staged data file**
* Although **read-only, external tables allow users to create virtual columns as expressions over the VALUE or METADATA column**
* **Querying data through external tables will likely be slower than native tables**
* They should be used to ***peek* at file contents or metadata rather than relying on them as data sources**
* Command used to create an external table: CREATE EXTERNAL TABLE <table-name>

###### ii) Directory Tables

* So, **external tables enable users to access staged file contents as if stored in regular tables**
* However, ***when only the file directory information is needed,* directory tables can be used**
* Conceptually, **directory tables resemble external tables**; however, there’re **several notable differences**:
* They are **NOT separate database objects** but **CAN be enabled as *part of* a stage**
* They **provide file metadata** such as path, size, + URL **but NOT file contents**
* They **do NOT allow the creation of virtual columns (or constraints)**
* They can be **created on *both* internal + external stages**
* **External tables permit users to access data stored in external cloud storage** using the same conventions as regular physical tables, but **directory tables only show file metadata**
* Directory tables are **more like the LIST function (returns a list of files from a stage) than an actual table**
* **Using a directory table instead of the LIST function allows users to use familiar SELECT syntax to access information**
* The following command displays how to create a directory table:
* CREATE STAGE <stage-name>

...

DIRECTORY = (ENABLE = TRUE);

#### Snowflake Views

* Snowflake **Views** behave like views in most RDBs, as in they **store a SELECT statement over physical objects as an object in a schema**
* Storing a SELECT statement as a shareable database object offers **several advantages to users of the system:**
* The **SELECT statement does not need to be written from scratch each time it is required**
* Provides **time savings + maintainability benefits through reusable modular code for data pipelines + analytics**
* There is **consistency in filters + business logic** for everyone with access to the view
* Views are **separate database objects** to the data sources they reference + **can therefore have different permissions that do not expose all the underlying data**
* They can **SELECT data from multiple sources at once**
* They **always shows the latest data** from source tables **without having to refresh it** (as opposed to **materializing** the results as a physical table)
* They have **zero storage cost as data is not physically copied but instead is read in real-time when the view is called**
* To create a view, use the following command: CREATE VIEW <view-name>
* These benefits come at the expense of **a single trade-off: performance**
* ***Each time* a view is called, the data must be *read* from the underlying tables, + all the logic and formulas in the view are calculated using compute resources + consume credits**
* However, **this is not as alarming as it sounds when we consider that Snowflake caching features also work for views**

##### a) Caching Views

* **Views attempt to take advantage of Snowflake’s various caches to mitigate performance + cost issues when executing a user query**
* The **results cache, managed by the services layer, holds all query results in the system, *including queries executed on views***
* ***As long as the underlying data has not changed*, subsequent queries will return the results instantly from the cache *if they are syntactically equivalent* without using compute credits**
* **If the results cache *cannot* be used, Snowflake will attempt to take advantage of the warehouse cache when referencing table data through a view**
* **Table data read from cloud storage will be available in memory through the warehouse cache for as long as the warehouse remains active, reducing the data retrieval times for the views that access it**

##### b) View Security

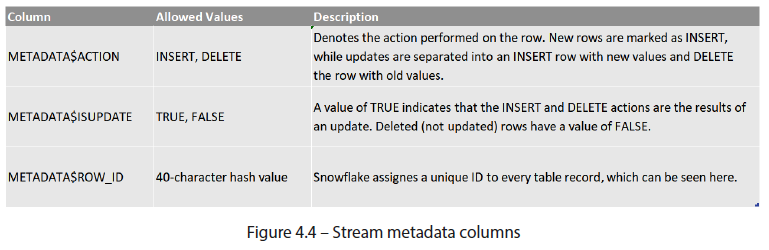
* As mentioned earlier, **views can be used to enforce data security as they are standalone securable database objects**
* **Tables containing sensitive data (at either the row or column level) can be restricted from public access through views**
* **Views can limit what columns are selected + what filters are applied + have separate access roles assigned to them**
* **Snowflake users have access to an added level of security by using secure views**
* A **secure view** (declared like a regular view, but with the word SECURE in the CREATE statement) functions like a non-secure (regular) view except that the **view definition (DDL) is NOT visible to the consumers, but only to the role that *owns* the view**
* **Some internal optimizations are disabled for secure views** to not expose the underlying data, but **performance is generally comparable to non-secure views**
* The following command creates a secure view: CREATE SECURE VIEW <view-name>

#### Materialized Views

* **Materialized views** are a Snowflake object that **offers benefits from both object types of tables *and* views**
* **Not exactly views, yet not quite tables,** they sit at the **intersection of these 2 object types** and **offer interesting benefits with one important caveat:**
* As the name suggests, materialized views are **actually *physical tables* that store the results of a view**
* **When the underlying data changes, the result is *re-materialized automatically***, which **means materialized views offer the performance of a table *and* a cached query result while offering all the security benefits of a regular view**
* The **only trade-off is the *cost:***
* Since materialized views **store query results in physical tables, they incur related storage costs**
* Additionally, **materialized views will use credits when refreshing the data**
* Materialized views have **certain limitations in the logic they permit in the SELECT statement compared to regular views:**
* Chief among them is that **materialized views can only be built over a *single* data source (stages are allowed), so they cannot use JOINs**
* This **reduces the use case of materialized views to that of pre-calculated table results or aggregates**
* On the plus side, the **Snowflake optimizer is smart enough to automatically redirect table queries to materialized views whenever possible**
* Snowflake users **should consider materialized views if *ALL* of the following conditions are met but discount them if *ANY* are not**:
* The **query results do not change often**
* The **view results are used significantly more frequently than data changes occur**
* The **query consumes a lot of compute resources**
* To create a materialized view, use: CREATE MATERIALIZED VIEW <view-name>

#### Streams

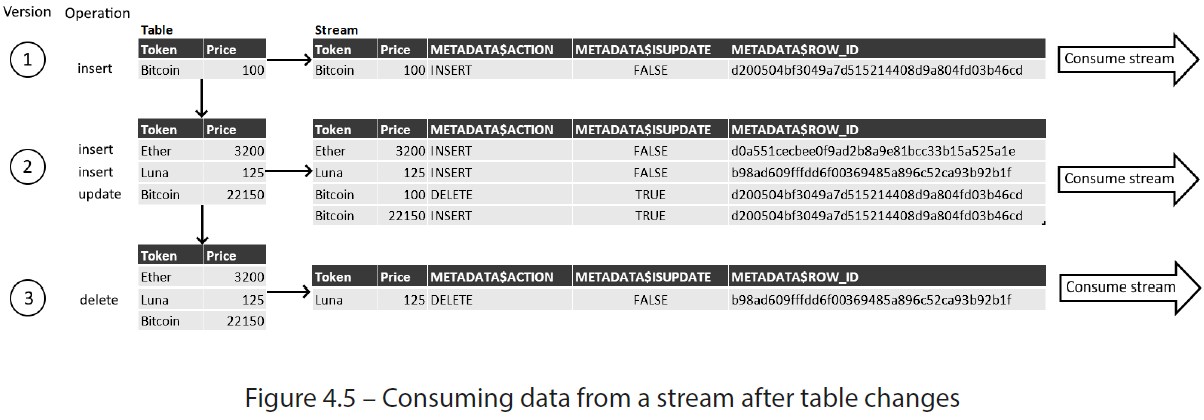
* **Streams** **are logical objects that capture data changes in underlying sources, including the previously mentioned objects (physical tables, views, and external + directory tables)**
* **Whenever a DML operation occurs in the source object, a stream tracks the changes (inserts, deletions, + the before/after images of updates)**
* Streams **achieve this through an offset storagetechnique by logically taking an initial snapshot of data + then tracking changes through metadata columns**
* Although **a stream can be queried *like* a table, it is NOT a separate object + does NOT contain table data**
* When **a stream is created, metadata columns are tacked onto the source object + begin tracking changes**
* The following table describes the metadata fields + their contents:



* This creates a stream on a table: CREATE STREAM <stream-name> ON TABLE <table-name>
* **For every *subsequent* DML operation, the stream holds the before + after images of the changed records and, by pairing them w/ the relevant metadata that identifies the type of DML operation (insert, update, or delete), enables accurate CDC to downstream objects**

##### Loading From Streams

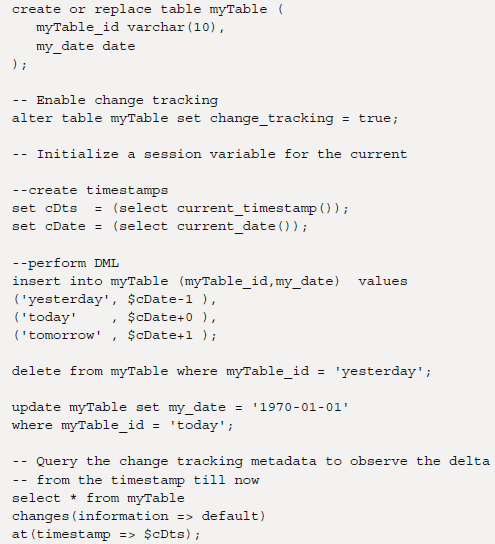
* **Snowflake keeps streams manageable + prevents them from growing excessively through an elegant method of consumption**
* **Consuming (using stream contents to (successfully) load or update downstream sources) empties the stream, readying it to absorb future DML changes w/out requiring manual periodic pruning**
* *Selects + failed consumption attempts will NOT clear the stream*
* In the following diagram, look at how this works in practice across various DML operations



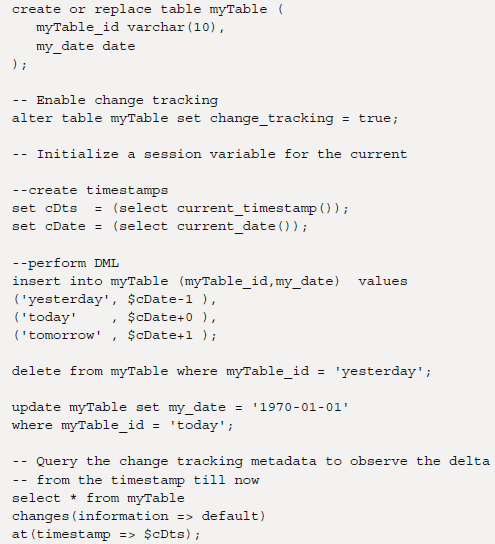
* *This example assumes the stream is consumed (emptied) after each change to the source table*
* *Depending on the nature of the data being tracked,* **streams are available in 2 varieties**:
* **Standard (or delta):** **Records inserts, deletes, + updates**
* Supported for tables, directory tables, + views
* **Append/insert-only**: **Only tracks *inserted* records + ignores any updates or deletions**
* Supported for physical tables, directory tables, + views as *append-only* and, for external tables, as *insert-only*
* **Being able to track changes w/out any processing overhead associated w/ manually tracking + comparing versions simplifies downstream DML operations by anticipating the type of change that needs to be updated**
* Due to this, **streams are frequently used when modeling fact tables + SCDs and in keeping them up to date**
* **Streams are intended to be consumed *regularly***
* Default retention period for changes in a stream = 14 days
* Although this can be extended, **Snowflake cannot guarantee stream freshness beyond that period**
* However, there’s **a** **less volatile alternative** **that doesn’t disappear when consumed:** **change tracking**, which can be used in conjunction w/ or as an alternative to streams for CDC

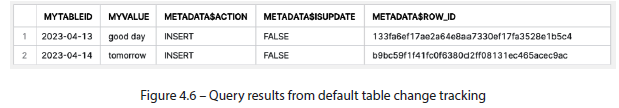
#### Change Tracking

* While a **stream is created as a standalone object** in a schema, **change tracking is enabled *directly* on tables, allowing Snowflake users to query CDC metadata**
* It **uses the same metadata fields found in streams but *appends them directly to a table***
* Unlike streams, the **changes are *NOT* eliminated if they are used to update downstream objects**
* Instead, the **change tracking persists for the data retention time of the table**
* Example of how change tracking is enabled + queried for a table where 3 records are inserted into a change tracking-enabled table, + subsequently, 1 record is updated + another is deleted:

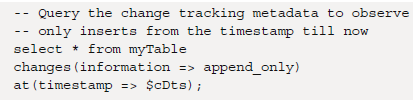


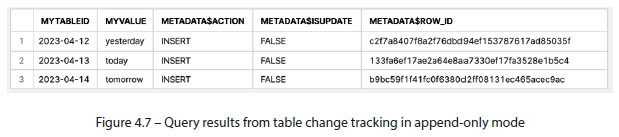
* The **result returned from querying the table’s change log shows the cumulative effect (latest version) of the record**
* Note that the yesterday and today records are absent from the result as they did not exist before the timestamp was set





* But **watch what happens when the *same* change log is queried in append-only mode:**





* The **result only returns records that were inserted into the table, regardless of subsequent deletions or updates**
* Notice that both the today and tomorrow records have returned while good day, which was an update, is absent
* **Barring some limitations, change tracking can even be enabled for views *as long as the underlying tables have it enabled as well***
* Since **change tracking** metadata persists for the *entire* data retention period of a table + cannot be cleared when consumed, it is **NOT suited for automated CDC pipelines but instead for analyzing the changes themselves**

#### Tasks

* Snowflake uses **Tasks** **to schedule + automate data loading + transformation**
* Although **data movement is not tracked in relational modeling, it is an integral part of transformational modeling and is covered here for completeness**
* **Tasks automate data pipelines by executing SQL in serial or parallel steps**
* They **can be combined w/ streams for continuous ELT workflows to process recently-changed table rows**
* **Can be done server-less-ly** (using auto-scalable Snowflake-managed compute clusters that do not require an active warehouse) **or using a dedicated user-defined warehouse**
* The code for creating a task is as follows:
* CREATE TASK <task-name>

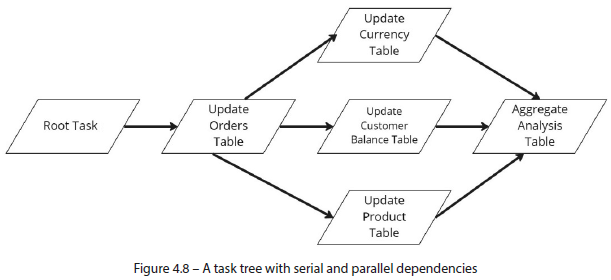
...

[ AFTER <parent\_task\_1> [ , <parent\_task\_2> , ... ] ]

[ WHEN <Boolean-expr> ]

AS <SQL>

* **Tasks are simple to understand: they run a SQL command (or execute a stored procedure) on a schedule or when called as part of a parent task**
* The following figure shows how tasks can be **chained serially + in parallel to form dependencies for data pipelines:**



* Stored procedures and scripting
* **Stored procedures** + Snowflake **scripting** allow you to **write procedural code using looping and branching constructs**, which falls outside the context of this book
* For information on working with **procedural logic**, refer to Snowflake documentation on
* Stored procedures (<https://docs.snowflake.com/en/sql-reference/stored-procedures.html>)
* Snowflake Scripting (<https://docs.snowflake.com/en/developer-guide/snowflake-scripting/index.html>)
* **Can take tasks even further by combining them with streams**

##### Combining Tasks and Streams

* **Tasks pair nicely w/ streams + allow users to *check for changes before executing***
* As part of their definition, **tasks have a WHEN <Boolean-expr> clause, which sets a logical condition that needs to be met before it can execute**
* The condition ***must* return a Boolean value but *can run any function or sub-select that the user wishes***
* Fittingly, **streams come w/ a system function that returns a value of TRUE whenever there are CDC records present +, otherwise, FALSE**
* The function can be called, referencing a stream name as follows:
* SYSTEM$STREAM\_HAS\_DATA('<stream-name>')
* **Setting this condition at the start of a task means it can be scheduled to run as frequently as changes are expected *but will only be executed when there are records to process***

#### Summary

* **Snowflake objects pack a lot of features, even behind familiar ones like tables + views**
* A **table** **in Snowflake can store more than just the data**
* **Depending on its settings, one can also hold months of historical + disaster recovery backups and offer offset change tracking for CDC**
* **Views**, likewise, **exceed expectations by providing change tracking + automated re-materialization**
* We saw how **Stages mark the entry point for data to make its way from external sources to Snowflake tables**
* Stages **also provide helpful features, such as external table access for reading file contents without copying them to a table beforehand**
* Finally, **to coordinate incoming data, establish automated ELT pipelines, + streamline CDC, Snowflake pairs tasks w/ streams to give its users full serverless or managed control (tying stages, tables, views, + all the connective transformational logic together)**

#### References

* Can refer to the following resources for additional information: *Understanding & Using Time Travel* (Snowflake documentation): <https://docs.snowflake.com/en/user-guide/data-time-travel.html>