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

## Part 3: Solving Real-World Problems with Transformational Modeling

### Chapter 13: Modeling Slowly Changing Dimensions

* In Chapter 7, we introduced to database **facts** + **dimensions**
* While **facts capture the *transactions* of business operations**, **dimensions help give those transactions *meaning* by providing descriptive attributes, groupings, + other contextual details**
* W/out careful curation + maintenance of dimension tables, databases are just facts, lacking all color and making meaningful analysis impossible
* **Dimensions shed light on the nature of entities in a data model, providing details** such as a customer’s billing address or a product’s description
* However, **entity details are constantly in flux in the fast-paced business world** (customers relocate and products gain new features)
* **A DW must be able to keep up w/ the steady stream of changes + allow users to quickly pivot between the latest state of the world + a *historical* perspective**
* Various dimension types are used to capture database entities’ slowly (or quickly) changing details
* *Leveraging unique Snowflake features discussed in previous chapters, we will create + update historical dimension details in cost-effective ways that have never been possible in other databases*
* Main topics:
* **Historical tracking** requirements in dimensional attributes
* The 7 types of **slowly changing dimensions** (**SCDs**)
* The structure and use cases of each SCD type
* Unlocking performance gains through Snowflake-native features
* Handling multiple SCD types in a single table
* Keeping record counts in dimension tables in check using **mini-dimensions**
* Creating **multifunctional surrogate keys** and comparisons with **hashing**
* Recipes for maintaining SCDs efficiently using Snowflake features

#### Technical Requirements for Local Snowflake Work

* The scripts used to instantiate + load the examples in this chapter are available in the following GitHub repo: <https://github.com/PacktPublishing/Data-Modeling-with-Snowflake/tree/main/ch13>
* While the key section of each script will be explained in the latter half of this chapter, refer to the repository for the complete code used to maintain + schedule the loading of the objects discussed

#### Dimensions Overview

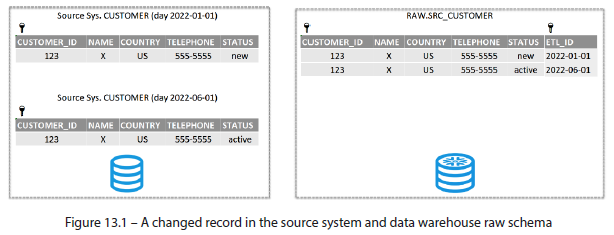
* A **dimension unifies (or conforms) similar attributes from one or various source systems into a *single* table under a common unique identifier known as a business key**
* A **single surrogate key can also be used in place of multi-column business or PKs**
* The **unique key of a dimension table plays a critical role in identifying dimension records + allows the database team to track + maintain changes over time**
* A **dimension table can be structured in predetermined ways to allow for different types of change tracking depending on the business requirement**

##### Slowly Changing Dimension Types

* **Attributes w/in a dimension have differing requirements for durability + change tracking**
* Some attributes are **updated directly**, while others require **historical snapshots**, yet others **cannot change at all (immutable)**
* This section will cover the types of **Slowly Changing Dimensions** **(SCDs , or update profiles)** a given attribute in a dimension can have
* It’s important to note that the **dimension type may not necessarily apply across all dimension attributes equally**
* **W/in the same dimension table, some attributes may be overwritten while others may not**
* By understanding SCD types + when to use them, database developers can implement the proper table structure + update techniques to satisfy the organization’s reporting + analytics requirements

##### Example Scenarios

* To explain the various SCD types, we use a simplified CUSTOMER dimension as an example + track the change as it would appear under each configuration
* Suppose our fact table stores order details from customer X, made on the 1st of every month in 2022
* Thanks to X’s patronage, their customer status went from “new” to “active” midway through the year
* Not only do we want to track *when* that change occurred, but we want to tie the *correct status* to the recorded sales facts (that is, the customer is “active” today, but half their orders were made as status “new”)
* The change in customer status is displayed here as it currently appears in the source system + DW landing area:



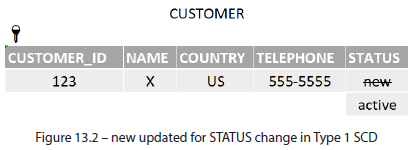
* With this scenario in mind, let’s explore the SCD types

###### Type 0 – Maintain Original

* Ironically, the first SCD **Type 0 does *NOT* change**
* **Type 0 dimensions are intended for *durable* attributes that cannot change due to their business nature**
* Examples of Type 0 attributes include birth **dates**, calendar dates, + any attribute recorded at record creation that needs to be tracked as a **baseline** (such as original price, weight, or date of first login)

###### Type 1 – Overwrite

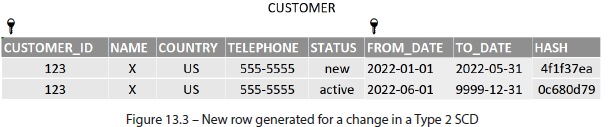
* **Type 1 attributes do not require historical tracking + may be directly overwritten w/ an UPDATE**
* Sometimes, the **latest attribute value is all the business cares about**
* Ex: Our organization demands the latest customer status, + previous values are irrelevant
* Maintaining a Type 1 dimension is relatively simple
* Ex: If the status changes, it is updated *directly* in the customer dimension, as illustrated here:



* However, **overwriting values is often not enough + a historical value must also be preserved**

###### Type 2 – Add a New Row

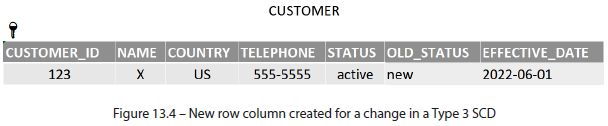
* For some attributes, an organization must **register the latest value + maintain prior historical records**
* **Type 2 attributes generate a new *row* every time a change is recorded**
* Generating new rows for a given business key **means uniqueness is violated unless a time dimension (the effective date) is added to the PK**
* **Effective date of a Type 2 SCD not only separates historical values for a given business key but also allows those records to be tied to fact tables at a given point in time**
* Maintaining a Type 2 SCD **requires creating new rows when record changes are detected and *additional* metadata columns to track them**
* A single record in our example would generate the following change in a Type 2 table:



* The following **metadata fields** make working w/ Type 2 attributes easier:
* **Validity intervals**: **B/c** **the business key is being duplicated w/ each change, *another* column must be added to the PK to maintain uniqueness**
* **Validity intervals** (also named valid\_from/to, start/end\_date) **provide the additional unique value for the PK + timestamp when the change occurred,** **allowing facts to be linked w/ the correct point-in-time dimension value**
* A **TO\_DATE column** also provides **a flag for identifying the latest record** using the standard surrogate high date of 9999-12-31
* **Hash**: **Using a hashing function, such as MD5, provides a quick + standard way to identify when record changes occur**
* This concept is borrowed from Data Vault (Chapter 17)
* **When there are *many* Type 2 attributes in a table, instead of checking for changes one by one, hash all of them into a single column + compare them in a single go**, as follows:
* Create the hash field: *SELECT MD5 (Col1 || Col2 || ... || ColN) AS hash*
* Compare the hash field: *IFF(hash\_new = hash\_old, 'same', 'changed')*

###### Type 3 – Add a New Column

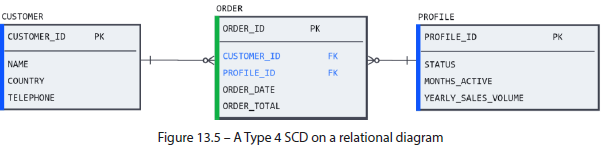
* **Type 3 dimensions** **track changes by adding a new column to store the previous value when a change occurs**
* The **original column is updated + *not* renamed to avoid breaking any existing downstream references**
* An **effective date metadata column records the time of the change, allowing analytics processes to use the new or historical value based on their validity period**
* An example of a status update in a Type 3 attribute is given here:



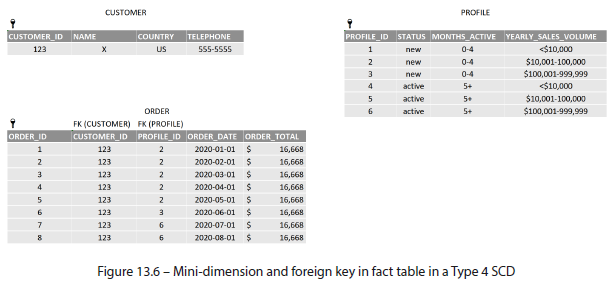
* Although **Type 3 is easier to maintain than Type 2, the limitation is storing *multiple* changes**
* While **Type 2 attributes can change as often as needed**, generating new rows each time, **Type 3 can only show one change w/out creating additional columns**, which is **not a scalable design if regular changes occur**

###### Type 4 – Add a Mini-Dimension

* When SCDs become ***quickly* changing dimensions** **(due to rapidly changing attributes**), **the number of records that Type 2 dimensions generate can cause performance issues**
* Especially true w/ dimensions containing many records (as in millions of rows or more)
* **Type 4** solution = to **split the frequently changing attributes into a separate** **mini-dimension**
* To **further curtail the number of records**, the **values in the mini dimension can be banded w/in business-agreed value ranges that provide a meaningful breakdown for analysis**
* The **mini dimension has its *own* surrogate key + does *not* contain the main dimension FK, allowing both to retain a relatively low cardinality**
* However, **to tie the main dimension to the mini, the mini-dimension FK *must* be included in the fact table (as the main dimension appears at the time of the generated fact)**
* On a diagram, the arrangement of a Type 4 dimension would look like this:



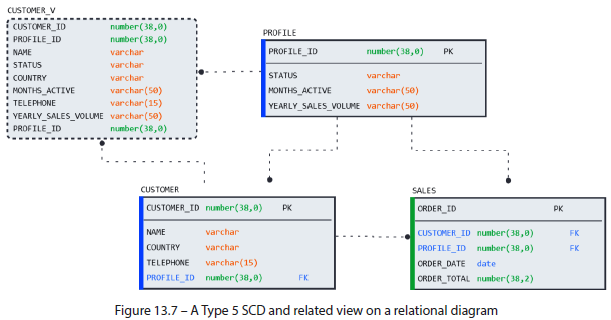
* For our example, the business wants to track the length in months for how long a customer has been active, as well as their total yearly spend at the time of each sale
* To avoid generating a record for each month + order placed, the business teams have agreed to group the MONTHS\_ACTIVE attribute into 2 categories (< 5 months or > 5 months) and to band the sales volume into 3 groups
* **The mini-dimension would need to contain every possible (or *allowable* by existing business rules) combination of groupings**
* Our example would look like this (notice how the profile ID changes throughout the year as a function of the customer’s attributes):



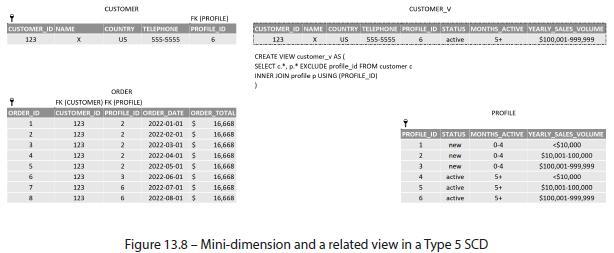
* While this arrangement satisfies the reporting requirement, **bridging dimension tables via a fact encumbers analysis *on the dimension itself***
* To **unify the main + mini-dimensions into *one*, a Type 5 SCD is used**

###### Type 5 – Type 4 Mini-Dimension and Type 1

* A **Type 5 SCD** is an extension of the Type 4 mini-dimension technique, **adding the mini-dimension key as a Type 1 attribute in the *main* dimension** (hence the name, 4 + 1 = 5)
* This approach **affords the performance gains of a Type 4 dimension by avoiding the explosive growth of rapidly changing Type 2 records + gives users a simple way to unify the main dimension w/ the mini dimension through a common JOIN column.**
* On a diagram, the arrangement of a Type 5 dimension would look like this:



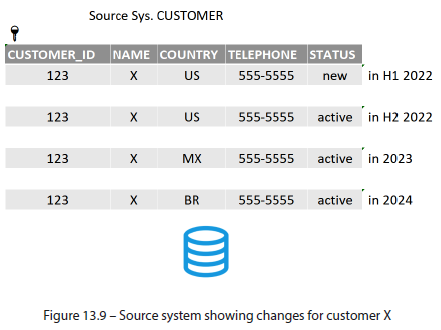
* Notice that **to further simplify the user experience, a *view* is created over the main *and* mini-dimensions to give the users a *single* entity to work w/**
* Analysis of the fact table becomes more versatile by allowing users to JOIN on *one* entity (the view) instead of the main + mini-dimensions if historical values are not required
* The same scenario described in the section on Type 4 would look like this under Type 5:



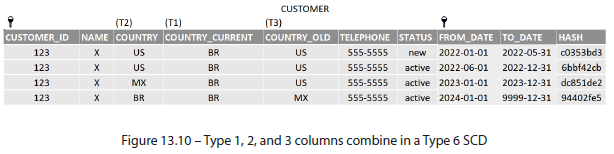
* **Unfortunately, Type 4, and by extension, Type 5, suffer from the inconvenience of calculating the mini-dimension value to include it as part of each fact**
* The **performance implications involved in adding the mini-dimension FK to the fact table should outweigh the performance gain in reducing the number of dimension records through the use of the mini-dimension**

###### Type 6 – Type 1 and 2 and 3 Hybrid

* **Type 6 SCD** is so named because it **combines the techniques of Type 1, 2, + 3** (1 + 2 + 3 = 6) **dimensions into one table**
* Based on business needs, users will demand different levels of historical values to achieve a balance of detail + flexibility in their analytics
* Suppose our customer *X* from previous examples began to relocate (moving HQ to Mexicoin 2023, then to Brazilin 2024)
* A **Type 6 approach yields a dimension table that gives analysts every possible temporal attribute value in every snapshot: a Type 1 current value, Type 2 effective dated value, + Type 3 previous value**
* To recap the status + country changes mentioned in this example, a snapshot of the source system over time is presented here:

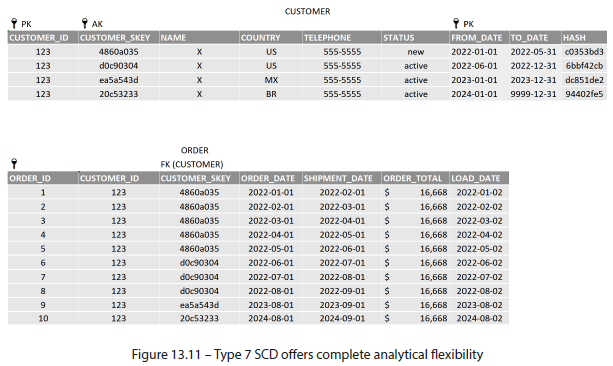


* In a business scenario where the customer status needed Type 2 *and* the country was presented as Type 1, 2, and 3, the resulting table would look like this (the hash column is now calculated as a function of status and country):

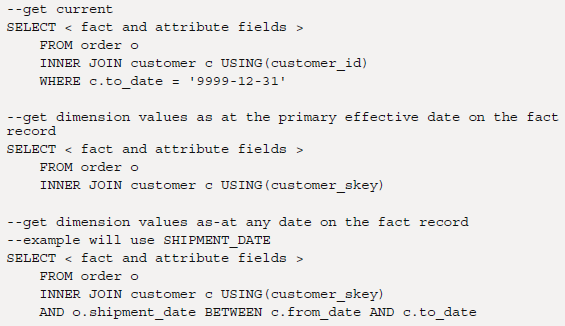


###### Type 7 – Complete As-At Flexibility

* Business users across all cultures + industries have a penchant for changing their minds
* The **Type 7 approach gives database modelers a way to deliver the needed historical attribute no matter the criteria or temporal reference point requested**
* A Type 7 dimension (unimaginatively named as the number that follows 6) **includes a natural key *and* a surrogate key in a Type 2 table structure + embeds both in the fact table**
* NOTE: A method for generating surrogate keys
* An efficient (and Data Vault-inspired) way to generate a surrogate key for Type 2 records is to use an MD5 hash on the compound PK (in this example, CUSTOMER\_ID and FROM\_DATE):
* *SELECT MD5(customer\_id || from\_date) AS customer\_skey*
* In a Type 7 configuration, a **surrogate key is added to an otherwise Type 2 structure + is embedded in the fact** (the latest SKEY as of the creation of each fact record)
* Based on the example scenario from the Type 6 section, the tables would look like this:

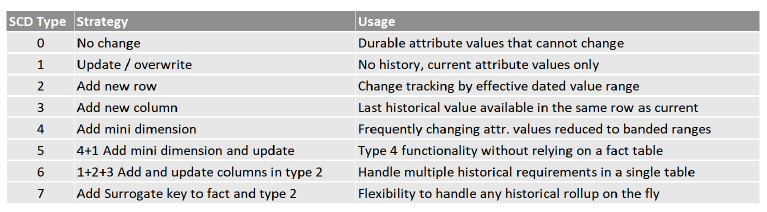


* A **Type 7 SCD allows business users to select the appropriate customer attributes based on the following criteria**:
* The **most recent or current information** (that is, TO\_DATE = '9999-12-31')
* The **primary effective date on the fact record** (that is, LOAD\_DATE)
* **When the user changes their mind, any date associated w/ the fact record** (that is, ORDER\_DATE or SHIPMENT\_DATE)
* Here is how those queries might look:



###### Overview of Slowly Changing Dimension Types

* While 8 (7 + Type 0) SCDs may seem like a lot, **most database designs rarely go beyond Type 3**, as the **first 4 SCD types strike an acceptable balance of performance, maintainability, + historical reporting needs**
* The following screenshot summarizes the 7 SCD types, including their maintenance strategy + usage

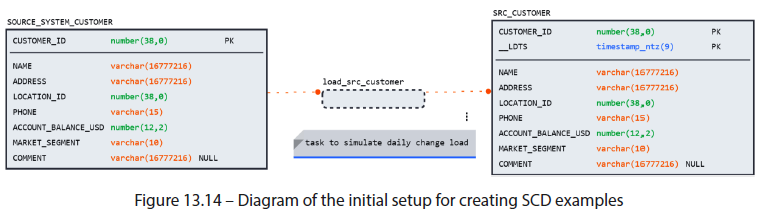


#### Recipes for Maintaining Slowly Changing Dimensions in Snowflake

* **Understanding the structure of an SCD + being able to load it correctly are very different concepts**
* W/ a firm grasp of SCD types, we will now cook up the recipes for creating + maintaining them in Snowflake
* Unlike generic SQL techniques you may have used in other databases, this book will take full advantage of the cost- and time-saving capabilities of Snowflake’s core features, such as **streams** and **zero-copy cloning**

##### Setting the Stage

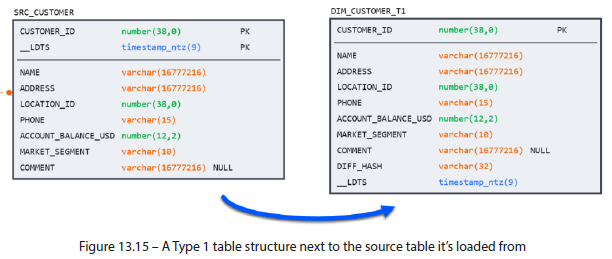
* To give readers complete autonomy to construct, experiment, + modify the upcoming exercises, we first **create a base table that will simulate the *day one* snapshot of the DW raw/source schema**
* The base table will **represent the initial *first* load of the source data into the DW**
* Next, we construct a **routine** that simulates a **daily load of new + changed records**
* For consistency w/ the 1st half of this chapter, these examples will use the CUSTOMER table from the *snowflake\_sample\_data.tpch\_sf10* schema
* Then, we create a **simulated SRC\_CUSTOMER table to represent the landing area of the DW** which, by default, will contain one quarter of the 1.5 million records of the sample CUSTOMER table
* Finally, we construct a **task**, LOAD\_SRC\_CUSTOMER, which will **randomly load 1,000 records** into the SRC\_CUSTOMER table (~80% new, ~10% modifications, + ~10% existing unchanged records)
* <https://docs.snowflake.com/en/user-guide/tasks-intro>
* The column that will receive changes in this example is account\_balance\_usd
* The parameters for the number of records loaded can be changed directly in the code
* Let’s recap the setup here:
* To get started, instantiate a new schema for running these examples, then create the 3 base objects as indicated in the accompanying code
* The file containing the examples is *create\_snowflake\_scd.sql*
* We also clone SRC\_CUSTOMER to create a backup for resetting + rerunning the examples:
* CREATE OR REPLACE SCHEMA ch13\_dims;
* CREATE OR REPLACE TABLE source\_system\_customer ... ;
* CREATE OR REPLACE TABLE src\_customer ... ;
* CREATE OR REPLACE TASK load\_src\_customer ... ;
* CREATE OR REPLACE TABLE src\_customer\_bak CLONE src\_customer;
* This script results in the following objects being created (the backup is not pictured):



* With the base objects in place, let’s begin with a Type 1 SCD

##### Type 1 – Merge

* The Type 1 table will have a similar structure to SRC\_CUSTOMER, + will even include the **metadata load date column, \_\_LDTS**
* However, *unlike* SRC\_CUSTOMER, which captures changes by *load date*, the Type 1 table will only have ***one* unique record for each entity in the dimension**
* For this reason, \_\_LDTS *cannot* be part of the PK but *will* be included as metadata to let users know the latest effective date of the record they are seeing
* Another field included in the Type 1 table is the DIFF\_HASH column
* Although the changes in our example only occur in one column, ACCOUNT\_BALANCE\_USD, using a DIFF\_HASH field can make equality comparisons faster, cleaner, and easier
* Create and populate the Type 1 table w/ the initial base load from SRC\_CUSTOMER by running the following statement: CREATE OR REPLACE TABLE dim\_customer\_t1... ;
* This results in the following table structure:



* Now, prime the SRC\_CUSTOMER table by calling the load task: execute task load\_src\_customer;
* Now, we are ready to perform the **update**
* **Updating Type 1 attributes requires a MERGE statement, which inserts new records *OR* updates changes**
* The key sections of the MERGE statement are highlighted here:
* MERGE INTO dim\_customer\_t1 dc

USING ( < SELECT latest snapshot from source > ) sc

ON dc.customer\_id = sc.customer\_id --unique identifier

WHEN NOT MATCHED --new records, insert

THEN INSERT VALUES ( < source columns >)

WHEN MATCHED --record exists

AND dc.diff\_hash != sc.diff\_hash –only update if changes exist

THEN UPDATE SET < target columns > = < source columns >

* **A MERGE statement is a relatively expensive database operation since it involves a join, a compare, *and* writing to disk in the form of inserts or updates**
* **To ensure we are not comparing records that have already been processed, include a filter that *only looks at source records that have not yet been processed*** (i.e. the latest *\_\_LDTS* value)
* In a typical DW scenario, this logic can be encapsulated in a view for ease of maintenance
* In this example, the logic has been embedded into the merge for ease of understanding:

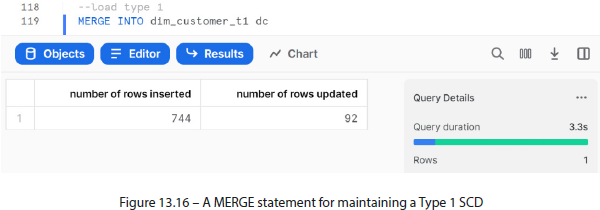
MERGE INTO dim\_customer\_t1 dc

USING (SELECT \*, MD5(account\_balance\_usd) AS diff\_hash

FROM src\_customer WHERE \_\_ldts =

(SELECT MAX(\_\_ldts) FROM src\_customer) ) sc

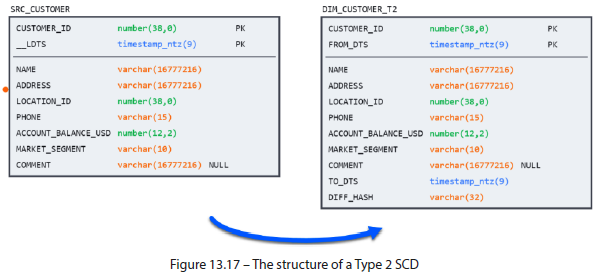
* Run the MERGE + observe the impact on the dimension table: MERGE INTO dim\_customer\_t1;
* As expected, of the 1,000 records loaded, ~3/4 were new records, + 10% were changes:



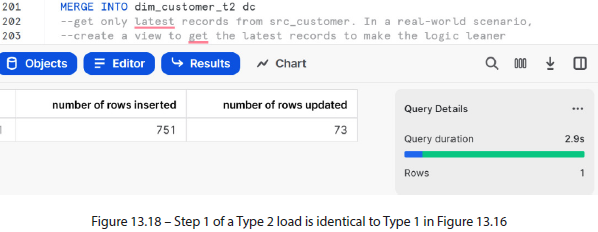
* Feel free to experiment by running additional loads + varying the number of records to see the impact on performance
* When ready, move on to Type 2

##### Type 2 – Type 1-Like Performance Using Streams

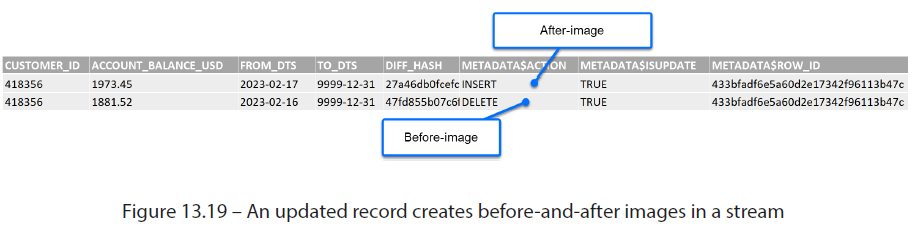
* **Type 2 tables are more performance-intensive than Type 1 because they contain historical changes and, over time, can grow to many times the size of the source table**
* However, in this section, we explore a technique that uses Snowflake **streams** **to achieve Type 1-like performance in a Type 2 load (via CDC)**
* *If unfamiliar w/ streams + the meta columns they contain, please revisit Chapter 4*
* And see <https://docs.snowflake.com/en/user-guide/streams-intro>
* They are **logical objects that capture data changes in underlying sources achieved through an offset storagetechnique by logically taking an initial snapshot of data + then tracking changes through metadata columns**
* Since we’ll be using the same base tables to perform a Type 2 load, *remember to reset SRC\_CUSTOMER to the original 375,000 rows by cloning it from the backup*, like so:
* CREATE OR REPLACE TABLE src\_customer CLONE src\_customer\_bak;
* Now, create + instantiate a Type 2 table by running: CREATE OR REPLACE TABLE dim\_customer\_t2;
* Recall this table contains metadata columns to track the validity of changed records over time
* The granularity of these validity intervals (Ex: monthly, daily, + millisecond) will depend on the data’s load frequency + reporting requirement
* Although Snowflake can maintain microsecond splits using the TIMESTAMP data type, **most reporting scenarios would not benefit from such near-real-time changes**
* Although daily changes (using the DATE data type) are the most commonly used, this example will use TIMESTAMP data types to allow users to run back-to-back loads on the same day
* The Type 2 table will look like this:



* **Note that only the FROM date column is required for the Type 2 PK**
* **In ensuring the data quality in a Type 2 setup, it is *essential* that for each BKEY that FROM dates are always unique, and FROM and TO intervals never overlap**
* Now, instantiate the **stream** for the Type 2 table + kick off a simulated source load to prepare for the MERGE statement:
* CREATE OR REPLACE STREAM strm\_dim\_customer\_t2 ON TABLE dim\_customer\_t2;
* EXECUTE TASK load\_src\_customer;
* ***Updating* a Type 2 table is done in two steps**
* 1) Run a MERGE statement
* 2) Update the changes like in a Type 1 load:
* MERGE INTO dim\_customer\_t2 dc;
* Notice here that the results and performance are identical to the Type 1 load so far:



* We’ve overwritten the current records w/ the latest values just like in a Type 1 load
* **Now comes the hard part: figuring out which *original* records were changed so that we can insert the before image and apply the correct FROM + TO dates**
* Luckily, **thanks to the previously created stream, this can be done without lookups or window functions,** **as the stream already contains the before-and-after images**!
* A sample record from the previous merge operation is displayed next
* Bear in mind that **under the hood, Snowflake is insert-only**, **+ it doesn’t delete or update the records *directly***
* Because of this, we can easily see the before image exactly as it appeared before the change:



* Knowing this, we can insert the before images of all the changed records into the table in a single operation, without performance-intensive joins or updates:
* INSERT INTO dim\_customer\_t2

SELECT < stream columns > FROM strm\_dim\_customer\_t2

WHERE metadata$action = 'DELETE'

* **After the insert, notice that the number of rows inserted matches the rows updated in the previous step**
* NOTE: 40% performance gain
* The **alternative to using streams for a Type 2 load is to use a temporary table**
* This approach, **although still requiring 2 steps, involves slightly more logic + suffers from the performance penalty of writing to 2 separate tables**
* However, to demonstrate the effectiveness of the streams technique, a comparison w/ a **dbt snapshot** **(snapshots are dbt’s version of a Type 2 SCD)** is included in the accompanying code
* On the standard load (using 1,000 records) the performance of both methods was identical
* However, when the record limit was removed + the full 1.5M rows were processed, the streams technique was 40% faster
* The dbt-generated DML + the results are included in the repository for this chapter
* **To simplify the daily loading activity of a Type 2 table, the 2-step loading process can be strung together as a series of sequential tasks that can be kicked off w/ a single command**
* The instructions for doing so are provided in the accompanying code
* Repeat the load operation using tasks, + when ready to move on to the Type 3 SCD, reset the SRC\_CUSTOMER table to baseline + continue

##### Type 3 – One-Time Update

* **Creating a Type 3 attribute, relatively speaking, is a simple + inexpensive operation**
* The process involves **altering the table to add an empty column + an update to set it equal to a base column**
* After that, **keeping the Type 3 table up to date is identical to the method used in Type 1**
* First, add the Type 3 column and set it to the baseline, like so:
* ALTER TABLE dim\_customer\_t3

ADD COLUMN original\_account\_balance\_usd number(12, 2);

UPDATE dim\_customer\_t3

SET original\_account\_balance\_usd = account\_balance\_usd;

* **W/ every insert, set the original column equal to the base column, + avoid updating it going forward**
* You can see the complete process in the accompanying code
* Having completed the exercise for a Type 3 dimension, we will wrap up the demonstration
* The remaining SCDs either combine the techniques used in Types 1, 2, + 3 or rely on straightforward modeling practices that rely on basic DML commands

#### Summary

* Due to the constantly changing nature of master data in the source system, the **DW must serve 2 critical functions to allow business users to pivot between current + historical attribute values in their reporting:**
* **1)** **Capturing source system changes in a landing area**
* **2)** **Creating SCDs that meet the organization’s reporting needs**
* **B/c master data plays such a key part in organizational analytics (often being tracked + scrutinized independently of fact records), learning to construct the required SCD structures + load them efficiently is a fundamental task for any DW team**
* We reviewed 8 different SCD structures for meeting various analytical needs: from durable Type 0 attributes that never change to dynamic Type 7 configurations that can handle any requirement
* Although many variations exist (even within SCD types), **Types 1-3 SCDs are the most often used as they strike an acceptable balance between maintainability, performance, + reporting requirements**
* Using the recipes provided in the accompanying SQL examples, this chapter explored the best practices for constructing SCDs by leveraging Snowflake-specific features such as **streams**, **cloning**, + **hashing**
* As demonstrated, **using native features such as streams can result in significant cost and performance savings compared to plain SQL methods**