Shwetank Singh
GritSetGrow - GSGLearn.com

DATA AND AI

SLOWLY CHANGING DIMENSIONS IN DATABRICKS

www.gsglearn.com

Slowly Changing Dimensions (SCD) in Databricks with PySpark and Delta Lake - By Shwetank Singh (GSGLearn.com)

Introduction

In data warehousing, **Slowly Changing Dimensions (SCD)** are techniques to manage dimensional data that changes slowly (e.g., customer addresses, product names) while preserving historical accuracy

Instead of simply overwriting data, SCD methods allow us to decide how to handle changes: keep old values, add new records, or retain limited history. Databricks, with **PySpark** and **Delta Lake**, provides an efficient platform to implement SCDs. Delta Lake brings ACID transactions and upsert capabilities (via *MERGE*), making it easier to handle dimension changes in one unified pipeline

In this document, we will explain different SCD types (Type 0, 1, 2, 3, 4, 6 and hybrid approaches), how they work, and demonstrate PySpark code snippets (using Delta Lake) for each. We will also use a sample dataset and a real-world scenario to illustrate these concepts.

Overview of SCD Types

There are multiple types of slowly changing dimensions, each with a specific strategy for handling changes:

- **Type 0 -- Fixed Dimension:** No changes are applied once recorded. The original value remains as-is (retain initial data)
- **Type 1 -- Overwrite:** Always update to the latest value. No history is kept; old data is lost
- **Type 2 -- Add New Row:** Add a new row/version for each change, preserving full history. Typically uses effective dates or flags to identify current vs historical records
- **Type 3 -- Add New Column:** Keep a limited history by adding additional columns to store previous values (e.g., "Previous Value") Only a few versions (usually one prior) are retained.
- **Type 4 -- Separate Historical Table:** Maintain a separate table for historical data, while the main dimension table holds only the latest version. This is also known as using a *mini-dimension* for rapidly changing attributes.
- **Type 6 -- Hybrid (Type 1 + 2 + 3):** A combination of Type 1, 2, and 3 techniques

  For example, a Type 2 dimension that also stores some overwrite-able current values or a "previous value" column (to enable both historical and current analysis in one place). Often implemented by adding both a history-tracking column and a current-value indicator to a Type 2 design.
- **Hybrid Approaches:** In practice, organizations may combine SCD strategies as needed. Some attributes might be treated as Type 1 while others are Type 2

within the same table (a mix-and-match approach). Advanced hybrid designs (Kimball's Type 5, 6, 7) provide additional flexibility by using mini-dimensions or dual records to allow reporting by original, historical, and current values

We will discuss examples of hybrid usage later.

Next, we'll set up a sample dimension dataset to illustrate how each SCD type is implemented using PySpark and Delta Lake.

Sample Dataset for Demonstration

Let's consider a simple **Customer Dimension** as our sample dataset. It contains the following fields:

- `CustomerID` -- the natural key for the customer (e.g., from source system).
- `Name` -- customer name.
- `City` -- customer city (this is an attribute that might change over time).
- `SignUpDate` -- the date the customer joined (an attribute that does not change after initial entry).

For example, our initial dimension table (current state) might look like this:

CustomerID	Name	City	SignUpDate
1	Alice	New York	2020-01-15
2	Bob	Los Angeles	2019-07-23

Now suppose we receive an update: Customer **Alice (ID=1)** has moved from **New York** to **Chicago** as of `2021-05-10`. We will demonstrate how each SCD type handles this change. In PySpark, we'll simulate the initial dimension as a Delta table and the incoming change as a source DataFrame.

First, let's create the initial Delta table for our dimension and the new data for the change:

```
`from pyspark.sql import functions as F

# Sample initial dimension data
data = [
    (1, "Alice", "New York", "2020-01-15"),
    (2, "Bob", "Los Angeles", "2019-07-23")
]
cols = ["CustomerID", "Name", "City", "SignUpDate"]
df_initial = spark.createDataFrame(data, schema=cols)

# Write initial dimension to a Delta table (e.g., path or table name)
dim_path = "/tmp/delta/customer_dim"
df_initial.write.format("delta").mode("overwrite").save(dim_path)

# New change record for CustomerID=1 (Alice moves to Chicago on 2021-05-10)
updates = [(1, "Alice", "Chicago", "2021-05-10")]
df_updates = spark.createDataFrame(updates, schema=cols)`
```

With this setup, we have a Delta Lake table at `dim_path` representing the current dimension, and a PySpark DataFrame `df_updates` representing the incoming changes. Next, we'll go through each SCD type and show how to process the change.

SCD Type 0 -- Fixed Dimension (Retain Original)

Definition: *Type 0* means **no changes** are applied to certain attributes at all -- the value remains as originally captured

This is used for attributes that should never change or where changes are ignored. For example, `SignUpDate` might be treated as Type 0 because once set, it should not be altered. If a change comes in for a Type 0 attribute, it is simply disregarded.

Implementation: In a Type 0 scenario, you do not perform any update on the attribute. The Delta Lake table is only initially populated, and further changes to Type 0 fields are ignored in the ETL logic. The code for Type 0 is trivial: you load the initial data and then do nothing for that field on updates.

PySpark/Delta snippet: For our example, if we consider `SignUpDate` as Type 0, we wouldn't update it even if a new value came in (which it shouldn't). The code would simply avoid including `SignUpDate` in any update statements. Here's how the initial load might look (which we already did above), and an illustrative approach to ignore changes:

```
`# (Already loaded initial data above for Type 0 fields; no update code needed for changes)
# If an update DataFrame has changes to a Type 0 field, we exclude that field from the merge or update.
from delta.tables import DeltaTable
deltaTable = DeltaTable.forPath(spark, dim_path)

# Example: we ensure SignUpDate is never overwritten. We do not include it in the update set.
deltaTable.alias("tgt").merge(
    df_updates.alias("src"),
    "tgt.CustomerID = src.CustomerID"
).whenMatchedUpdate(set={
    # Update other fields like City, Name, but exclude SignUpDate
    "Name": F.col("src.Name"),
    "City": F.col("src.City")
    # SignUpDate omitted to retain original
}).whenNotMatchedInsert(values={
    "CustomerID": F.col("src.CustomerID"),
    "Name": F.col("src.Name"),
    "City": F.col("src.City"),
    "SignUpDate": F.col("src.SignUpDate")  # Insert uses original signup date
}).execute()`
```

In the above merge, if a matching customer is found, we update the Name and City but leave `SignUpDate` untouched (not listed in the `set` clause, so it remains as in `tgt`). Thus, any incoming change to `SignUpDate` would be ignored. In practice, you might also simply not allow `SignUpDate` to appear in the source updates for existing

customers at all. After the merge, for Customer 1, `SignUpDate` stays `"2020-01-15"` (original) even if the source had a different value.

Type 0 is the simplest SCD type -- **no history is tracked because the attribute never changes by design**. This approach is suitable for immutable attributes like an original registration date or an initial category that should never be modified.

SCD Type 1 -- Overwrite (Latest Value)

Definition: *Type 1* SCD overwrites the old data with the new data, keeping only the latest state. No historical data is stored -- once an attribute changes, the prior value is lost

. This method is used when you don't need history, only the current truth (for example, correcting a spelling mistake in a name, or updating a phone number where only the latest contact matters).

Behavior: Using our example, under Type 1, when Alice's city changes to Chicago, we simply update her City from "New York" to "Chicago" in the customer dimension, and we **do not keep** the old city anywhere in the dimension. After the update, the dimension will show Alice as living in Chicago, and there's no trace in the dimension table that she ever lived in New York.

Implementation in Delta Lake: We can use Delta Lake's `MERGE` operation to perform this upsert (update insert) efficiently. The merge will find the matching dimension row by `CustomerID` and update the City. If the incoming record was new (no match), it would insert it as a new customer. Delta Lake handles this in one transaction (atomic), which is simpler and safer than manual upsert logic

PySpark Code for SCD Type 1:

```
`from delta.tables import DeltaTable

deltaTable = DeltaTable.forPath(spark, dim_path)
# Perform Type 1 merge (overwrite changes, no history)
deltaTable.alias("tgt").merge(
    df_updates.alias("src"),
    "tgt.CustomerID = src.CustomerID"
).whenMatchedUpdate(set={
    # Overwrite with latest values
    "Name": F.col("src.Name"),
    "City": F.col("src.City"),
    "SignUpDate": F.col("src.SignUpDate")  # typically unchanged, but if it were allowed to change, Type1 would overwrite
}).whenNotMatchedInsert(values={
    # Insert new record if not present
    "CustomerID": F.col("src.CustomerID"),
    "Name": F.col("src.Name"),
    "City": F.col("src.City"),
    "SignUpDate": F.col("src.SignUpDate")
}).execute()`
```

After this merge, the Delta table is updated in-place. In our scenario, Customer 1's City would now be "Chicago". If we query the dimension table:

`spark.read.format("delta").load(dim_path).show()`

We would see Alice's city as Chicago, and Bob remains Los Angeles. The old city "New York" is no longer in the table. Type 1 is straightforward and useful when **historical reporting is not required**. Its advantages are simplicity and low storage usage (since we don't add rows), but the disadvantage is obvious: **loss of history** -- we cannot see past values after updates.

SCD Type 2 -- Add New Row (Full History)

Definition: *Type 2* is the most commonly used SCD technique for preserving full history

When a change occurs, *a new row* is added to the dimension table with the updated data, and the old row is marked as historical (no longer current). This allows unlimited history tracking -- every change results in a new version in the dimension. To manage these versions, Type 2 dimensions use additional fields, typically:

- A **surrogate key** (unique primary key for each dimension row version).
- **Valid-from and valid-to dates** (or start and end timestamps) to indicate the period during which each version was active.
- A **current flag** (boolean or indicator) to easily filter the current record per natural key.

With these, one can join fact data to the dimension version that was current at the fact's date, enabling historically accurate analysis.

Behavior: In our example, under Type 2, when Alice moves to Chicago, we **insert a new record** for Alice with the new city. The existing Alice record (with New York) is marked as no longer current (for instance, set `IsCurrent = false` and `EndDate = '2021-05-10'` which is the day before the change became effective, assuming the change effective date is 2021-05-10). The new record for Alice in Chicago gets `IsCurrent = true` and a `StartDate = 2021-05-10`. We also assign a new surrogate key to this new record (if using surrogate keys). All new facts after May 10, 2021 will now point to the new surrogate key, thus linking to "Alice in Chicago". Facts before that date remain linked to the old surrogate key ("Alice in New York")

. This way, queries can correctly aggregate data by the city Alice was in at the time of each fact.

Implementation in Delta Lake: Implementing Type 2 in Delta can be done with a combination of updates and inserts. Delta Lake's **MERGE** can handle this in one operation by using conditions: update the old record and insert the new record. We'll need to incorporate the SCD metadata columns. Let's alter our dimension schema to include `StartDate`, `EndDate`, and `IsCurrent`. We initialize existing records with `StartDate` as their original date (SignUpDate in this case), `EndDate` as a high-date or NULL for current, and `IsCurrent = true`. For convenience, let's assume we are adding these columns now (in a real scenario, you'd design the dimension with these columns from the start for Type 2).

We will use the merge on condition `CustomerID` and `IsCurrent=true` (to target only the current version of each customer). The logic will be:

- When matched (i.e., the current record exists) and a change in City is detected, update the existing record to set `EndDate` and `IsCurrent=false`.

- When not matched (no current record for that CustomerID, or after updating the old one), insert the new record with the updated City, `StartDate` as the change date, `EndDate = NULL` (or a future end like 9999-12-31), and `IsCurrent=true`.

PySpark Code for SCD Type 2:

```
`from pyspark.sql.functions import lit, expr, current_date

# Ensure the Delta table has SCD Type 2 columns (StartDate, EndDate, IsCurrent).
# For demonstration, alter the existing data:
deltaTable = DeltaTable.forPath(spark, dim_path)
# Let's update the existing records to fit Type 2 structure (initial load
adjustments).
deltaTable.update(
    condition="true",  # update all rows
    set={
        "StartDate": F.col("SignUpDate"),            # use SignUpDate as the start
date for initial load
        "EndDate": lit(None),                       # no end date yet (current record)
        "IsCurrent": lit(True)                      # mark all initial records as
current
    }
)`
```

Now perform the merge for the incoming change:

```
`# Merge for Type 2 SCD
deltaTable.alias("tgt").merge(
    df_updates.alias("src"),
    "tgt.CustomerID = src.CustomerID AND tgt.IsCurrent = true"
).whenMatchedUpdate(condition="tgt.City <> src.City", set={
    # expire the old record
    "EndDate": lit("2021-05-09"),       # one day before change date, or use
expr("date_sub(src.SignUpDate, 1)")
    "IsCurrent": lit(False)
}).whenNotMatchedInsert(values={
    # insert the new record with updated info
    "CustomerID": F.col("src.CustomerID"),
    "Name": F.col("src.Name"),
    "City": F.col("src.City"),
    "SignUpDate": F.col("src.SignUpDate"),        # carry original signup or use src
date as needed
    "StartDate": lit("2021-05-10"),              # the effective date of this change
    "EndDate": lit(None),                       # open-ended for current record
    "IsCurrent": lit(True)
}).execute()`
```

(In a real ETL, you might use `current_date()` or a timestamp from the source to set Start/End dates. Here we hardcoded `"2021-05-10"` for clarity.)

After this merge, our Delta dimension table would have **two records for CustomerID=1**:

- **Alice -- New York** with `StartDate = 2020-01-15`, `EndDate = 2021-05-09`, `IsCurrent = false` (historical record).

- **Alice -- Chicago** with `StartDate = 2021-05-10`, `EndDate = NULL`, `IsCurrent = true` (current record).

Customer 2 (Bob) remains unchanged with his single current record.

We have effectively preserved history: if we query the table for Customer 1, we see both the old and new city with their date ranges. Querying only current customers is easy (`WHERE IsCurrent = true`). To find what city Alice lived in on e.g. 2020-12-01, we can filter date >= StartDate and (date <= EndDate or EndDate is null) etc.

Delta Lake's ACID capabilities ensure this change (update + insert) happens atomically. Also, Delta supports **Time Travel**, so even without SCD columns, one could query older table versions. However, SCD Type 2 stores history *as data*, making it straightforward to join with fact tables on surrogate keys and analyze historical data at scale

Note: Each Type 2 change should generate a new surrogate key in a dimensional model. In our code above, we used `CustomerID` as the key just for illustration. In practice, you might have a surrogate key column (e.g., `CustomerKey`) that is unique per row. Delta Lake can auto-generate surrogate keys or you handle it in the ETL (e.g., by keeping a sequence or using identity columns in Delta if available).

SCD Type 3 -- Add New Column (Limited History)

Definition: *Type 3* SCD captures a limited history (usually just the previous value) by adding extra column(s) for the prior value of an attribute

Instead of adding new rows on changes, we add (one-time) new columns such as "PreviousCity". When a change occurs, we shift the current value into the "previous" column and then update the current column with the new value. This way, we retain the old value alongside the new one in the **same row**. However, this only keeps track of a fixed number of changes (often just one prior value). If the attribute changes again, you would lose the oldest value unless you had multiple columns (PreviousCity2, etc.), which gets unwieldy. Type 3 is useful when you only care about comparing the current value to the immediate previous value (for example, *current department* and *previous department* of an employee).

Behavior: In our example, to implement Type 3 for the City attribute, we would alter the Customer dimension to have a new column, say `PreviousCity`. Initially, this might be null or same as current for the first load. When Alice's city changes from New York to Chicago, we update Alice's single dimension row: set `PreviousCity = "New York"` (the old city), and set `City = "Chicago"` (the new city). We do **not create a new row**. The dimension now tells us Alice's current city and her previous city. If another change comes (say Alice moves to Boston later), a pure Type 3 design would then lose the fact she was in New York: we'd set PreviousCity to Chicago and City to Boston (New York would be gone). Thus, only one historical step is kept.

Implementation: For Type 3, since we only update a single row (no insert of new rows), the implementation is similar to Type 1 but with an additional column update. We need to update two columns at once: the current value and the "previous" value. Delta Lake supports this with `UPDATE` or `MERGE` operations. We'll assume our dimension now has a `PreviousCity` column (added beforehand, default null or could be filled with initial City for convenience).

PySpark Code for SCD Type 3:

First, let's alter the table to add a PreviousCity column (for demonstration, we'll do this via DataFrame for simplicity):

```
`# Add PreviousCity column to the Delta table (for demo, do it via DataFrame
transformation)
df_dim = deltaTable.toDF()
# Initialize PreviousCity as null for all
df_dim = df_dim.withColumn("PreviousCity", lit(None))
# Overwrite the delta table with this new schema (in a real scenario, use Delta's
schema evolution or DDL)
df_dim.write.format("delta").mode("overwrite").save(dim_path)
deltaTable = DeltaTable.forPath(spark, dim_path)`
```

Now perform the Type 3 update for the incoming change (Alice's city):

```
`# Prepare source with the new city (already in df_updates)
# Merge to update City and PreviousCity
deltaTable.alias("tgt").merge(
    df_updates.alias("src"),
    "tgt.CustomerID = src.CustomerID"
).whenMatchedUpdate(condition="tgt.City <> src.City", set={
    "PreviousCity": F.col("tgt.City"),   # store old city
    "City": F.col("src.City")           # update to new city
}).whenNotMatchedInsert(values={
    "CustomerID": F.col("src.CustomerID"),
    "Name": F.col("src.Name"),
    "City": F.col("src.City"),
    "SignUpDate": F.col("src.SignUpDate"),
    "PreviousCity": lit(None)           # new customers have no "previous" city
}).execute()`
```

After this merge, the dimension table still has one record per customer. For Customer 1 (Alice), the row now looks like:

- `CustomerID=1, Name=Alice, City=Chicago, PreviousCity=New York, SignUpDate=2020-01-15`

Bob's record remains the same (PreviousCity is null or not applicable). We have effectively kept Alice's old city in the `PreviousCity` column. If we needed to know the last city she lived in before the current one, it's directly available.

Type 3 is beneficial for tracking *one prior state* without the complexity of multiple records. It's easy to query (no need to join or filter for current vs historical---both values are in one row)

However, it **only preserves limited history**

. If requirements grow to track more changes, Type 3 falls short. In practice, Type 3 is used sparingly (for example, to track a previous customer segment or last job title, while full history might not be needed).

SCD Type 4 -- Separate Historical Table (Mini-Dimension)

Definition: *Type 4* uses **two tables** to manage changes

. The main dimension table holds the current version of each entity (one row per business key for the latest state), and a separate **history table** (or mini-dimension) stores all the historical changes. This approach is useful when an attribute changes very frequently or keeping all history in the main dimension would cause performance or storage issues. By offloading historical versions to a separate table, the main dimension stays slim and fast for current lookups, while history can be queried from the archival table as needed. Kimball calls one flavor of this a *mini-dimension* (especially if you break off a set of volatile attributes into another dimension).

Behavior: In our example, if City were highly volatile and we chose a Type 4 approach, we would have:

- **DimCustomer_Current** (the main dimension) containing the latest city for each customer (and no other city history). E.g., after Alice moves, this table would simply show Alice in Chicago.
- **DimCustomer_History** (the historical table) containing records of changes, possibly with columns CustomerID, OldCity, NewCity, ChangeDate (or a structure similar to Type 2 with StartDate/EndDate for each city version). Each time a customer's city changes, we would *insert a record* into the history table capturing that change, and also update the main table. For instance, when Alice moves to Chicago, we'd record a history entry: (CustomerID=1, City was "New York", changed to "Chicago", ChangeDate=2021-05-10). The main table is then updated to "Chicago". If another move happens, another history row is added (previous "Chicago" to new city, etc.), and main table updated again.

Implementation: Implementing Type 4 involves two operations when a change is detected:

1. **Insert into History Table:** Write a record of the old state (and new state, or just old state with effective dates) into the historical Delta table.
2. **Update Main Table:** Overwrite the main dimension's current value with the new value (Type 1 update on main).

We'll assume we have two Delta tables: `dim_customer_current` at `dim_path` (main table) and `dim_customer_history` at `history_path` (history table). The history table could have columns like `CustomerID, City, StartDate, EndDate` or a pair of City columns to indicate change from -> to. Here, for simplicity, we log each change as a separate row with the old city and the date it ended.

PySpark Code for SCD Type 4:

```
`# Paths for main and history tables
current_path = "/tmp/delta/dim_customer_current"
history_path = "/tmp/delta/dim_customer_history"

# Ensure the update to main is reflected (Type 1 update on current table)
currentTable = DeltaTable.forPath(spark, current_path)

# Identify changes by joining update with current table
updates_df = df_updates.alias("src").join(
    currentTable.toDF().alias("tgt"),
    F.col("src.CustomerID") == F.col("tgt.CustomerID")
).filter(F.col("src.City") != F.col("tgt.City")).select(
    F.col("src.CustomerID").alias("CustomerID"),
    F.col("tgt.City").alias("OldCity"),
    F.col("src.City").alias("NewCity"),
```

```
    F.col("src.SignUpDate").alias("ChangeDate")  # reuse SignUpDate field for change
date here
)

# Insert old state into history table for each change
if updates_df.count() > 0:
    # Prepare history entries (OldCity and when it ended)
    history_entries = updates_df.withColumn("EndDate", F.col("ChangeDate"))
    history_entries = history_entries.select("CustomerID", "OldCity", "EndDate")
    history_entries.write.format("delta").mode("append").save(history_path)

    # Update main current table with new city (Type 1 update for current data)
    for row in updates_df.collect():
        currentTable.update(
            condition=f"CustomerID = {row['CustomerID']}",
            set={"City": F.lit(row["NewCity"])}
        )`
```

In this snippet, we:

- Determined that Customer 1 has `OldCity = "New York"` and `NewCity = "Chicago"`.
- Appended a row to the history table: (CustomerID=1, OldCity="New York", EndDate="2021-05-10"). This records that New York was Alice's city up until that date.
- Then updated the main table so Alice's City becomes "Chicago".

After these steps:

- **DimCustomer_Current** will have Alice in Chicago, Bob in Los Angeles.
- **DimCustomer_History** will have a record of Alice's old city. If Bob had a change, it would have similar entries.

Type 4 keeps the main dimension very compact (only current data). Historical analysis is a bit more involved since you'd have to consult the history table (or combine it with current for a full view). In some cases, Type 4 is implemented by designing a mini-dimension for a frequently changing subset of attributes and linking that mini-dimension to the fact table separately But the essence is separating current vs history into two locations.

With Delta Lake, both tables can be maintained easily. The history table can grow large, but since it's not joined to facts on every query (only when historical reporting is needed), this can mitigate performance issues of a huge Type 2 dimension.

SCD Type 6 -- Hybrid (Type 1 + Type 2 + Type 3)

Definition: *Type 6*, also known as *Hybrid SCD* or "1+2+3", combines aspects of Type 1, Type 2, and Type 3

The goal of Type 6 is to enjoy the benefits of full history (Type 2) while also easily accessing current values and perhaps a previous value in the same record (Type 1 and Type 3). In practice, a Type 6 dimension is often a Type 2 dimension that includes additional columns for current values or previous values that are updated along with new row inserts.

For example, you might have a Type 2 dimension with `StartDate`, `EndDate`, `IsCurrent`, and also a column `PreviousCity` (Type 3 style) and possibly even a duplicate of the current attribute that gets overwritten across all histories (Type 1 style for a particular attribute). This sounds complex, but it provides flexibility: one can analyze facts by the attribute value that was in effect at the time (using the Type 2 historical records), or by the current value of that attribute (using a Type 1 style attribute that was overwritten on all records, or via a separate reference). **Kimball** describes Type 6 as "Add Type 1 attributes to a Type 2 dimension" -- essentially storing a continuously updated current value in each historical record for easy comparison

Behavior: Suppose we want to design a Customer dimension that allows analysts to query facts by both **the city at the time of the fact** and **the customer's current city** without complex joins. We could implement Type 6 as follows:

- Maintain Type 2 history of City (each change = new row, with Start/End dates, etc.).
- Add a `CurrentCity` column in the dimension. When a customer's city changes, we create a new row (Type 2) and also update **all** rows of that customer to set `CurrentCity = new city` (that's a Type 1 aspect applied to the `CurrentCity` column across historical rows). Additionally, we might keep a `PreviousCity` column in the new row as in Type 3 (the last city before the change).

After such an update: the new row will show the new City, the previous city in `PreviousCity`, and its `CurrentCity` (which might be same as City for the new row). All older rows for that customer would have their `CurrentCity` updated to the latest city as well, even though their own `City` field (historical) remains unchanged. This way, any row in the dimension can tell you "what is the current city of this customer" (even for old records), enabling comparisons like "was the sales made when Alice lived in New York higher or lower than her sales now that her current city is Chicago?" without a self-join. Type 6 essentially embeds a point-in-time value and the current value in each record

Implementation: Implementing full Type 6 in code can be involved, since it requires multiple actions: insert new row, update old row(s) and possibly update some fields in all historical rows. However, let's illustrate a simplified hybrid approach: we will combine Type 2 and Type 3 (and a bit of 1). We'll maintain the `PreviousCity` as in Type 3 and `IsCurrent` with dates as in Type 2. We won't update all old rows' current city for brevity (that would be an extra step where we run an update on all rows of that CustomerID to set a `CurrentCity` field).

So, our dimension structure for Type 6 might be: `CustomerID, Name, City, PreviousCity, StartDate, EndDate, IsCurrent`. The process for a change:

- Mark the existing record as not current (EndDate set).
- Insert the new record with the new City, and set its `PreviousCity` to the old City (carry forward the immediate last value).
- (Optionally, update a `CurrentCity` field on all records, but we'll skip this part in code).

PySpark Code for SCD Type 6 (Hybrid example):

```
`# Assuming deltaTable is our dimension with PreviousCity, StartDate, EndDate,
IsCurrent columns (similar to after Type 2 + Type 3 setup)
# We will do a two-step approach for clarity: expire old and collect old value, then
```

```
insert new.

# 1. Expire the current record (Type 2 part)
deltaTable.alias("tgt").merge(
    df_updates.alias("src"),
    "tgt.CustomerID = src.CustomerID AND tgt.IsCurrent = true"
).whenMatchedUpdate(condition="tgt.City <> src.City", set={
    "EndDate": lit("2021-05-09"),
    "IsCurrent": lit(False)
}).execute()

# 2. Insert new record with PreviousCity (Type 3 part) using the old value.
# Get the old city value from the previously current record (now expired) for use as
PreviousCity
old_city_df = spark.read.format("delta").load(dim_path).filter(
    "CustomerID = 1 AND IsCurrent = false AND EndDate = '2021-05-09'"
).select(F.col("City").alias("OldCity")).limit(1)

# Prepare new record DataFrame
new_rec = df_updates.join(old_city_df)  # join to get OldCity for CustomerID=1
new_rec = new_rec.withColumn("PreviousCity", F.col("OldCity"))\
                 .withColumn("StartDate", lit("2021-05-10"))\
                 .withColumn("EndDate", lit(None))\
                 .withColumn("IsCurrent", lit(True))

# Append the new record to the Delta table
new_rec.select("CustomerID","Name","City","SignUpDate","PreviousCity","StartDate","EndDa

      .write.format("delta").mode("append").save(dim_path)`
```

In step 1, we used a merge to set the existing Alice record `IsCurrent=false` and `EndDate=2021-05-09`. In step 2, we built a new record for Alice in Chicago: we looked up her `OldCity` (New York) from the now-expired record and set that as `PreviousCity` for the new row, with `StartDate=2021-05-10`. We then appended this new row. Now our dimension has:

- Alice (New York, PreviousCity=null, StartDate=2020-01-15, EndDate=2021-05-09, IsCurrent=false)
- Alice (Chicago, PreviousCity=New York, StartDate=2021-05-10, EndDate=null, IsCurrent=true)

This is very similar to the pure Type 2 result, except we have a PreviousCity on the new row explicitly storing "New York". If we had also maintained a `CurrentCity` in every row and updated it across the board, that would complete a classic Type 6 (so that even the first row might have `CurrentCity=Chicago` after the change, reflecting the latest info). That could be done with an additional update if needed.

Type 6 (and other hybrid strategies) provide **more analytic options** at the cost of more complexity

. In many real-world scenarios, a simpler approach (like pure Type 2) is sufficient, but Type 6 can be handy if you need to easily compare historical facts with current attributes in one dimension join. Delta Lake can handle the required multiple updates/inserts, but careful orchestration (or use of Delta Live Tables with

appropriate configurations) is needed to ensure all steps occur atomically and consistently.

Hybrid Approaches in Practice

Beyond the formal types above, **hybrid approaches** often refer to tailoring the change tracking to specific business needs. A common hybrid approach is to use *different SCD types for different columns in the same dimension*. For example, a customer dimension might treat `Address` changes as Type 2 (new row for each address change, to retain full history), but treat `PhoneNumber` as Type 1 (overwrite, since keeping old phone numbers might not be necessary), and treat `SignupDate` as Type 0 (never changes once set). This mix ensures history is preserved where it matters and avoided where not needed. Such a design is essentially a combination of SCD strategies, sometimes informally called **Type 7** or others in literature, but the key is that it's driven by the requirements for each attribute.

The Kimball Group has also defined specific hybrid types beyond Type 6: for example, **Type 5** (Type 4 + Type 1 outrigger) and **Type 7** (dual Type 1 and Type 2 dimensions)

These are advanced designs to handle very specific reporting needs (like allowing easy access to current values alongside historical ones through separate dimension links). In practice, these are less common and can usually be achieved via simpler means or additional columns.

In summary, hybrid SCD implementations give more flexibility for analysis -- you can choose to preserve history for some attributes and not for others, or provide multiple ways to view data (historical vs current) in your dimensional model. Delta Lake's support for complex merge conditions and transactions makes implementing even complex hybrids feasible. However, it's wise to balance complexity with actual business needs; sometimes a straightforward Type 2 with some Type 1 columns is enough, rather than implementing every hybrid twist.

Real-World Use Case Example

Use Case: Consider a retail company managing a **Product Dimension** in a Databricks Lakehouse. Each product has attributes like *Name*, *Category*, and *Price*. Business requirements dictate that they need to track changes in **Category** over time (to analyze historical sales by the category a product was in at the time of sale), but they do not need to keep old prices in the dimension (because the sale facts already capture the price paid, or they always want reports to use the latest price). This scenario calls for a hybrid SCD approach: use Type 2 for Category, and Type 1 for Price.

Solution: They implement the product dimension as a Delta table with SCD Type 2 structure (surrogate key, effective dates, etc.) for category changes. When a product changes category, a new dimension row is inserted (new surrogate key, new category, current flag true) and the old row is expired -- preserving the old category history. However, for price updates, instead of adding new rows, they simply overwrite the `Price` in the current dimension record (Type 1 behavior), since historical prices aren't required in the dimension. In other words, the ETL logic checks which attribute changed:

- If **Category** changed, do a Type 2 merge (insert new row, expire old).
- If only **Price** changed (category same), do a Type 1 update on the existing row (no new row).

- If both changed, it would do a Type 2 (new row) to preserve category history, and the new row will reflect the new price; the old price isn't kept as a separate history because they don't need it. The old row is expired with its old category, and its price value as it was (which they decided not to track separately).

They also maintain a column `IsCurrent` and dates in the dimension for Category. For querying current data, they simply filter `IsCurrent=true`. For historical analysis, they join facts on product surrogate keys with the appropriate effective date range (or use the surrogate directly if the fact stores the surrogate foreign key at time of transaction).

Delta Lake Implementation: Using PySpark, they leverage Delta Lake merges to handle this logic. A simplified outline of their merge logic could be:

```
`productTable = DeltaTable.forPath(spark, "/path/to/product_dim")
changes = spark.read.format("delta").load("/path/to/product_changes")  # prepared
change set with new category/price

productTable.alias("tgt").merge(
    changes.alias("src"),
    "tgt.ProductID = src.ProductID AND tgt.IsCurrent = true"
).whenMatchedUpdate(condition="src.Category = tgt.Category", set={
    # Category didn't change, so just update current fields (Type 1 for Price or
others)
    "Price": F.col("src.Price"),
    "Name": F.col("src.Name")
}).whenMatchedUpdate(condition="src.Category <> tgt.Category", set={
    # Category changed -> expire old record (Type 2)
    "EndDate": F.col("src.ChangeDate"),
    "IsCurrent": lit(False)
}).whenNotMatchedInsert(values={
    "ProductID": F.col("src.ProductID"),
    "Name": F.col("src.Name"),
    "Category": F.col("src.Category"),
    "Price": F.col("src.Price"),
    "StartDate": F.col("src.ChangeDate"),
    "EndDate": lit(None),
    "IsCurrent": lit(True)
}).execute()`
```

In the above pseudo-code:

- If categories are the same (only price or name might have changed), it does a simple update in place (Type 1 update for those attributes).
- If category changed, it updates the existing row to set it as not current (end dating it), and the insert clause will add the new product row with the new category and current price. (We used two `whenMatchedUpdate` with different conditions --- Delta's extended merge syntax allows multiple whenMatched clauses with conditions, evaluated in order).

Outcome: This hybrid SCD implementation allows the company to query the Product dimension in two ways. If they join facts to the product dimension on the surrogate key used at the time of the sale, they'll get the category as it was at that time

(because the fact would link to the correct historical row by surrogate key). If they want the *current* category of products for all sales (for example, reclassify historical sales under the new categories), they could join on `ProductID` and pick the dimension row with `IsCurrent=true` (or maintain a separate current dimension table if needed, akin to Type 4 for category). With Delta Lake handling the merges, the nightly update process is robust --- it can insert and update in one job, and the ACID guarantees mean analysts won't see partial updates. This approach has been running in production, and thanks to Delta Lake's performance, even thousands of product changes per day are processed efficiently (no full table scans or overwrites --- just merges on the keys and relevant partitions).

Conclusion: In this document, I've explored various Slowly Changing Dimension types and how to implement them using PySpark and Delta Lake on Databricks. Each SCD type serves a purpose, from never changing (Type 0) to full audit history (Type 2) to hybrid combinations for complex needs. Delta Lake's ability to perform upserts and maintain data versioning makes it an ideal choice for handling these dimension changes efficiently. By choosing the right SCD strategy (or combination) for your dimensions, you can ensure your data warehouse preserves the necessary history and provides correct data for all time periods with optimal performance.

THANKS - Shwetank Singh - GSGLearn.com