Managing Data Changes with SCDs in Databricks

Written By Francisco Maver

Why are Slowly Changing Dimensions important?

Even with all the hype around real-time streaming, **Slowly Changing Dimensions (SCDs)** are still crucial for building data systems you can actually trust. SCDs preserve the historical context, which is key when tracking changes in products, customers, or organizational hierarchies, all essential capabilities for modern analytics.

When you track how your data changes over time, you unlock some pretty important capabilities. You can stay compliant with regulations like GDPR and CCPA because you have accurate records of past customer states. Your historical analysis becomes way more reliable since you're anchoring metrics to the right dimensional version instead of getting misleading insights. For regulated industries, this auditability is non-negotiable (you need verifiable, time-aware records). And for data governance, understanding when and how your data changed makes debugging and lineage tracking so much easier.

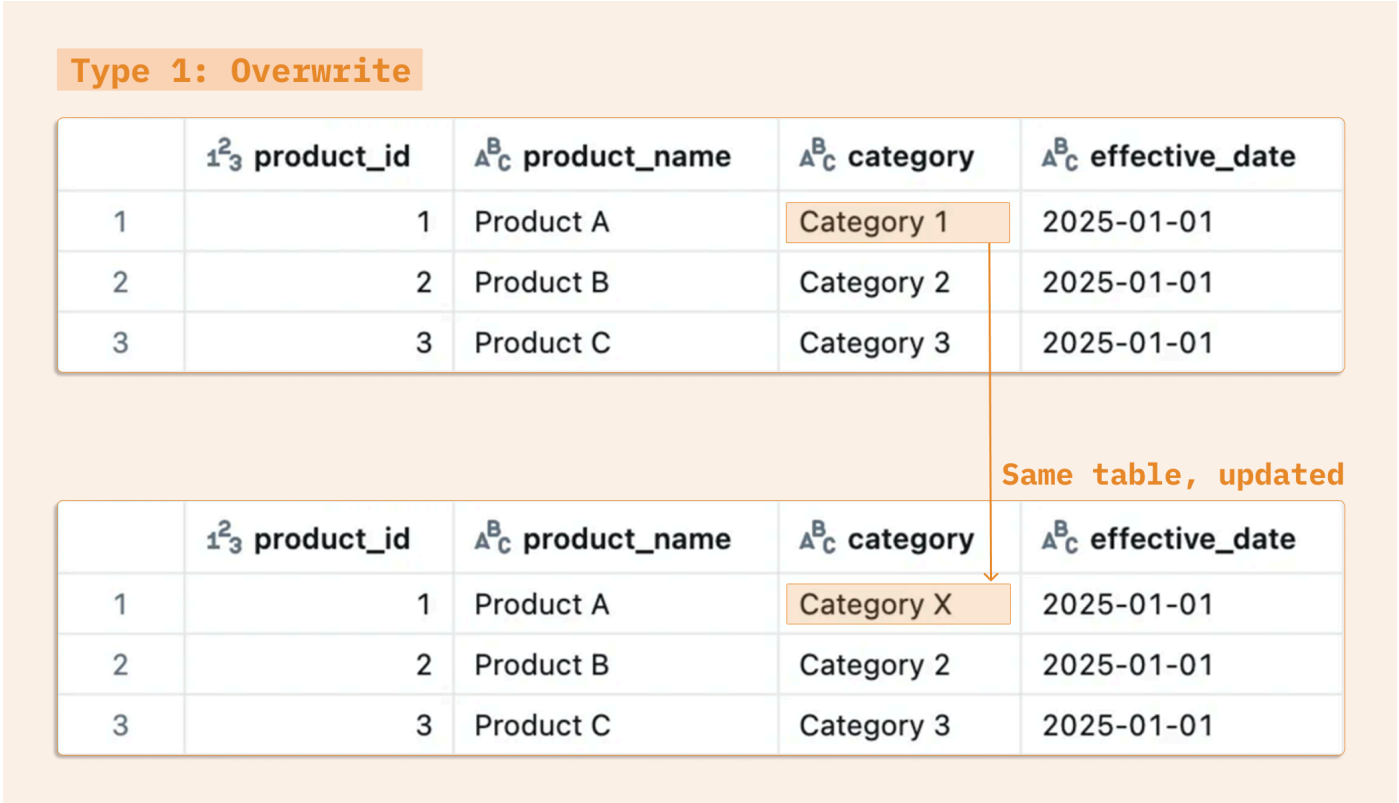
Source: SunnyData

and Python APIs, leveraging the platform's native features for robust dimensional change tracking. Let's start with the basics and break down the different SCD types.

Types of Slowly Changing Dimensions (SCD)

Type 0 (Retain Original) **maintains the original values without tracking any changes**. This works for dimension attributes that should never change, such as a customer's original registration date or a date of birth.

Type 1 (Overwrite) **historical values are simply overwritten with new values**. Although this approach is straightforward and requires minimal storage, it does lose historical tracking. Type 1 is good for correcting inaccurate data or for attributes where historical values aren't important.



If the category for **product_id = 1** changes and we don't track history, the old value is simply overwritten.

Type 2: Add New Row

	product_id	product_name	category	effective_date	start_date	end_date	current_flag
1	1	Product A	Category 1	2025-01-01	2025-01-01	2025-06-24	0
2	1	Product A	Category X	2025-06-25	2025-06-25	null	1
3	2	Product B	Category 2	2025-01-01	2025-01-01	null	1
4	3	Product C	Category 3	2025-01-01	2025-01-01	null	1

To preserve history, we keep old records defining between which dates was active

*For the *end_date* column, there are 2 possibilities for current data:

- Keep them `null`
- Put a big future date such as `9999-12-31`

Using a future end_date like '9999-12-31' is preferable to NULL. This approach simplifies date comparisons and makes time-based filters and joins more straightforward.

Type 3 (Add New Attributes) tracks limited history by **adding new columns to store previous values**. For example, a dimension might have "Current Category" and "Previous Category" columns. This approach offers a middle ground between Types 1 and 2 but **only preserves one historical state**.

3	3	Product C	Category 3	2025-01-01	null

Type 4 (History Table) uses **separate tables for current and historical data**. The main dimension table contains only current values, while a history table stores all previous versions. This approach optimizes query performance for current data while maintaining complete history.

Type 5 SCD **combines Type 1 and Type 4** approaches. It keeps current values in a main table while storing complete history in a separate table, optimizing both query performance and historical tracking. Type 5 is **ideal for high-volume environments** where most queries need current data, but historical accuracy remains essential for compliance or analysis.

SCD **Type 6 combines elements of Type 1, 2, and optionally 3.** This allows access to *current values*, *historical versions*, and *previous states* in one single record. This level of flexibility can be useful in **highly specialized reporting** scenarios, but for many teams, it may add **unnecessary complexity** compared to more common patterns like SCD2 alone.

If you're planning to implement Type 6, it can be done in different ways depending on the architecture. In this blog, we focus on the **single-table approach**, which combines current, previous, and versioned fields into one unified schema. However, it's also common to see **two-table implementations**, where one table holds current values (Type 1) and another stores full historical records (Type 2/3).

Type 6: Hybrid Approach: Current + Previous + Versioning

	product_id	product_name	category	effective_date	start_date	end_date	current_flag	original_category
1	1	Product A	Category 1	2025-01-01	2025-01-01	2025-06-24	0	Category 1
2	1	Product A	Category X	2025-06-25	2025-06-25	null	1	Category 1
3	2	Product B	Category 2	2025-01-01	2025-01-01	null	1	Category 2
4	3	Product C	Category 3	2025-01-01	2025-01-01	null	1	Category 3

historical record

current record

```
ON target.product_id = source.product_id -- Your business key
WHEN MATCHED THEN
  UPDATE SET
    target.product_name = source.product_name,
    target.category = source.category,
    target.effective_date = source.effective_date,
    target.last_updated = current_timestamp() -- Always update the latest
WHEN NOT MATCHED THEN
  INSERT (product_id, product_name, category, effective_date, last_updated)
  VALUES (source.product_id, source.product_name, source.category, source.effective_date, c
```

```
# Assume 'bronze_products' is your daily snapshot of customer data
# Assume 'dim_products_scd2' is your SCD Type 2 dimension table

from delta.tables import DeltaTable
```

```
    .where("updates.effective_date > destination.effective_date")
    .where("destination.current_flag = 1")
)

# STEP 2: Stage updates for the merge
# - Updated records to insert (new version)
# - Existing records to be expired (will match by product_id)
staged_updates = (
    updated_records_to_insert.selectExpr("NULL as mergeKey", "updates.*")
    .union(
        bronze_products.selectExpr("product_id as mergeKey", "*")
    )
)

# STEP 3: Perform the SCD2 merge
# - Expire previous records (set current_flag = false and set end_date)
# - Insert new records (new version or new product)
destination_table = DeltaTable.forName(spark, "my_catalog.my_schema.dim_product_scd2")

destination_table.alias("destination").merge(
    source=staged_updates.alias("staged_updates"),
    condition="destination.product_id = staged_updates.mergeKey"
).whenMatchedUpdate(
    condition="""
        destination.effective_date < staged_updates.effective_date
        AND destination.current_flag = 1
    """,
    set={
        "current_flag": "false",
        "end_date": "staged_updates.effective_date"
    }
).whenNotMatchedInsert(
```

```
        "start_date": "staged_updates.effective_date",
        "end_date": lit("9999-12-31") # or null
    }
).execute()
```

```
# Read the streaming CDC source (products table)
@dlt.view
def products():
  return spark.readStream.table("cdc_data.products")

# Declare the target streaming table
dlt.create_streaming_table("products_scd1")

# Auto-manage the SCD Type 1 merge
dlt.create_auto_cdc_flow(
  target = "products_scd1",
  source = "products",
  keys = ["product_id"],
  sequence_by = col("sequence_num"),  # ensures correct ordering
  apply_as_deletes = expr("operation = 'DELETE'"),
  apply_as_truncates = expr("operation = 'TRUNCATE'"),
  except_column_list = ["operation", "sequence_num"],
  stored_as_scd_type = 1
)
```

Francisco Maver

https://www.linkedin.com/in/fmaver/