# Slowly Changing Dimensions, Temporal Modeling, and Master Data Versioning for Data Governance

## Introduction: Overview of SCD and Master Data Versioning

In the realm of data warehousing and analytics, managing evolving business information is a critical challenge. Slowly Changing Dimensions (SCD) are foundational techniques for tracking changes in master and reference data over time. Coupled with robust master data versioning and temporal modeling, SCD methods enable organizations to ensure data reliability, regulatory compliance, and effective dispute resolution. This article provides a comprehensive technical review of SCD Types 0-6, temporal tables, bitemporal modeling, key management strategies, and governance-driven implementation patterns. It is designed for data engineers, architects, and analysts seeking authoritative guidance on data change management and governance.

## Technical Deep Dive: SCD Types 0-6

### SCD Type 0: Retain Original

SCD Type 0 preserves the original dimension values indefinitely, disallowing any updates. It is suitable for immutable attributes such as birth date or legal entity creation date. This type provides a static reference point, supporting regulatory requirements where historical accuracy is paramount.

### SCD Type 1: Overwrite

Type 1 SCD involves overwriting dimension values upon change, with no history retained. It is best applied when historical changes are irrelevant for analytics, such as correcting a misspelled name. While this method minimizes storage and processing requirements, it does so at the expense of auditability.

### SCD Type 2: Add New Row (Full History)

Type 2 SCD creates a new record for each change in the dimension, capturing complete change history. Each row includes effective dates and, optionally, version numbers. This method is essential for regulatory compliance, audit trails, and point-in-time analysis, supporting rollback and dispute resolution.

### SCD Type 3: Add New Attribute (Partial History)

Type 3 SCD stores previous values in additional columns, typically tracking only a limited number of historical states (e.g., current and previous address). This approach is useful for attributes where only recent changes matter for analysis or reporting.

### SCD Type 4: History Table

Type 4 SCD maintains a current dimension table and a separate history table. The history table preserves all changes, supporting complex auditing and rollback scenarios. This pattern is often chosen when performance or storage constraints preclude full history in the main dimension table.

### SCD Type 5: Hybrid (Mini-Dimensions)

Type 5 combines Type 1 and Type 4, using a mini-dimension to track frequently changing attributes separately while maintaining a current dimension table. This hybrid approach optimizes performance and storage while providing adequate historical tracking for volatile attributes.

### SCD Type 6: Combined Approach

Type 6 SCD integrates Types 1, 2, and 3, storing current and historical values in the same row along with effective dates and versioning. This approach supports flexible querying, point-in-time analysis, and governance requirements for both full and partial history.

## Temporal Tables and Bitemporal Modeling

Temporal tables, available in modern relational databases, automatically track historical changes to data by maintaining valid-from and valid-to timestamps. Bitemporal modeling extends this concept by supporting both business time (when data is valid in the real world) and system time (when it was recorded in the database). This dual tracking is essential for organizations subject to regulatory scrutiny, enabling “as of” queries and accurate reconstruction of data states for any point in time.

## Surrogate vs Natural Keys

Surrogate keys are artificial, system-generated identifiers (typically integers) used to uniquely identify dimension records, independent of business logic. Natural keys, on the other hand, are derived from business attributes (such as Social Security Number or Customer Code). Surrogate keys are preferred in SCD implementations as they allow seamless management of historical records and avoid complications from changes to natural key values. Natural keys are often retained for reference and integration but should not be used as primary identifiers in evolving master data.

## Implementation Patterns: Change Tracking and Audit Trails

Robust SCD implementations require precise change tracking and comprehensive audit trails. Key implementation patterns include:

- Effective date fields for validity periods
- Version numbers for explicit record ordering
- User and timestamp metadata for auditing changes
- Automated triggers or stored procedures for consistent change capture

Audit trails support regulatory compliance and dispute resolution by enabling reconstruction of data states and change history.

## Master Data Versioning: Metadata, Policies, Retention, Purging, and Conflict Resolution

### Version Metadata Fields

Master data versioning relies on metadata fields such as version number, effective date/time, expiration date/time, change reason, and author. These fields facilitate granular tracking and querying of changes.

### Versioning Policies

Organizations must define clear policies for when new versions are created (e.g., on every change, only on material changes, or at scheduled intervals). Policies should align with business requirements and governance standards.

### Retention and Purging

Retention policies determine how long historical versions are preserved. Regulatory requirements often mandate multi-year retention. Purging strategies should balance compliance, storage costs, and business need, with archived copies retained for audit purposes when necessary.

### Conflict Resolution

Version conflicts may arise in distributed or multi-user environments. Common resolution strategies include last-write-wins, manual review, or merging changes based on business rules. Proper conflict management is critical for master data integrity.

## Governance and Compliance: Why SCD Matters

SCD and master data versioning are vital for effective data governance. They provide:

- Regulatory Compliance: SCD enables organizations to meet legal requirements for data retention, auditability, and traceability, especially in finance, healthcare, and public sectors.
- Dispute Resolution: Historical data facilitates investigation and resolution of business disputes by reconstructing the state of data at any given time.
- Impact Analysis: Change tracking allows organizations to assess downstream effects of data modifications, supporting change management and risk mitigation.
- Rollback Capability: SCD Type 2 and temporal tables enable reversal of erroneous changes, safeguarding data integrity.

## SCD Type Selection: Governance Decision and Use Cases

The choice of SCD type is a governance decision balancing auditability, performance, and business requirements:

- Type 1: Use when only the current value is relevant and historical changes are not needed (e.g., correcting typos).
- Type 2: Essential for full historical tracking, regulatory compliance, and rollback scenarios (e.g., customer address history).
- Type 3: Appropriate when only recent changes are needed for analysis (e.g., tracking prior and current sales region).

Effective governance mandates a formal evaluation of these tradeoffs, documented in data management policies.

## Temporal Queries for Governance

Temporal tables and SCD implementations support advanced governance queries:

- Point-in-Time Reporting: Retrieve the state of master data as of a specific date, supporting regulatory and audit requirements.
- 'As Of' Joins: Join fact and dimension tables based on effective dates, ensuring accurate historical analysis.
- Change Frequency Analytics: Analyze how often key attributes change, informing governance and data quality initiatives.

These capabilities underpin robust data governance and enable organizations to meet evolving compliance standards.

## Conclusion: Best Practices and Recommendations

Managing slowly changing dimensions and master data versioning is foundational for data governance, regulatory compliance, and business resilience. Best practices include:

- Use surrogate keys for dimension tables to ensure robust historical tracking.
- Implement SCD Type 2 or temporal tables where full history and rollback are required.
- Define clear versioning, retention, and purging policies aligned with governance objectives.
- Leverage metadata fields and audit trails for comprehensive change tracking and dispute resolution.
- Regularly review SCD type selection to align with evolving business and regulatory requirements.

By adopting these strategies, data engineers, architects, and analysts can ensure that their data environments are resilient, compliant, and equipped for effective governance.

**DB instance identifier- database-scd-type2**
**Master username- postgres**
**Master password**-Achieve123*

he special features of a **Delta Table** (used in your PySpark script) compared to the **standard PostgreSQL tables** (created in your RDS query) center around how they handle data integrity, history, and scale.

Here are the key differences:

## 1. Native Time Travel (Versioning)

This is the most significant difference for your hands-on project.

- **Standard Table:** To "rollback" data, you had to manually write a complex `rollback_version` procedure that finds old data and re-inserts it.
- **Delta Table:** It has an automatic **Transaction Log** (Delta Log). You don't need a manual procedure; you simply use `versionAsOf` to see the table exactly as it was at any point in time.

## 2. ACID Transactions

While PostgreSQL is ACID compliant, Delta Lake brings these "database-grade" features to **Data Lakes** (like S3 or HDFS).

- **Atomic:** When you run the `merge` in PySpark, either the entire SCD Type 2 update succeeds or none of it does. You won't end up with "half-updated" data if the script crashes.
- **Consistency:** It ensures that multiple users can read and write to the same table simultaneously without seeing corrupted data.

## 3. The "Merge" Command vs. Procedural Loops

- **Standard Table:** Your PostgreSQL code uses a `FOR` loop to check every row one by one and decide whether to expire or insert. This is slow for millions of rows.
- **Delta Table:** It uses a **declarative MERGE statement**. This allows Spark to distribute the work across many computers, making it significantly faster for Big Data.

## 4. Schema Evolution

- **Standard Table:** If you want to add a new column (like `customer_email`), you must manually run `ALTER TABLE` commands.
- **Delta Table:** It supports **Schema Evolution**. You can simply add a setting to your PySpark script to automatically update the table structure when the incoming data changes.

## 5. Audit Capability

- **Standard Table:** You had to create a separate `scd_version_audit` table and write code to manually log actions into it.
- **Delta Table:** It has a built-in `DESCRIBE HISTORY` command. This automatically logs who changed what, when, and with which command (e.g., MERGE or DELETE), providing an instant audit trail without extra code.

DEMO:

Here is a total explanation of your SCD Type 2 implementation, comparing the PostgreSQL (Traditional RDBMS) approach and the Delta Lake (Modern Data Lake) approach.

---

## 1. The Core Concept: What is SCD Type 2?

Slowly Changing Dimension (SCD) Type 2 is a method used to track the **history** of data changes over time. Instead of overwriting an old address or name, we "expire" the old record and "insert" a new one.

### The Three Pillars of our Implementation:

- **Version Numbering:** Every change gets a new version ID (V1, V2, V3).
- **Current Flag (`is_current`):** A boolean that tells the user which record is the "truth" right now.
- **Effective Dates (`effective_to`):** A timestamp marking when a record stopped being active.

---

## 2. Comparing the Two Implementations

### A. The PostgreSQL Approach (Traditional)

In your PostgreSQL demo, you likely used standard SQL `INSERT` and `UPDATE` statements within a transaction.

- **Mechanism:** Manual logic checks if a record exists. If it has changed, you issue an `UPDATE` to the old row and an `INSERT` for the new one.
- **Rollback:** If you need to roll back, you manually insert a "Version 3" that looks like "Version 1" to keep the forward-only audit trail.

### B. The Delta Lake Approach (Modern)

In your PySpark/Delta demo, you utilized **Upsert (Merge)** and **Time Travel**.

- **Mechanism:** We use a single `MERGE` operation. The "Merge with Join" logic allows us to expire and insert rows in one atomic transaction.

- **Time Travel:** Unlike PostgreSQL, where you have to manually recreate old rows, Delta Lake's **Transaction Log** remembers the physical state of the files at every version.
- **Rollback:** You demonstrated two ways to roll back:
    1. **Physical Restore:** Using `restoreToVersion(0)` to physically reset the table to its original state.
    2. **Audit Rollback:** Appending the old data as a new Version 3, mimicking the PostgreSQL style while maintaining the Data Lake's history.

---

## 3. Demo Step-by-Step Walkthrough

When you present your demo, show these four specific states:

### State 1: Initial Ingest

- **Action:** Load `initial_data.csv`.
- **Result:** Alice is in New York, `is_current` = true, `version_number` = 1.

### State 2: The Update

- **Action:** Load `update_data.csv` (Alice moves to London).
- **Result:** You now see **three rows**.
    - Row 1: Alice (New York) | `is_current` = **false** | `effective_to` = [Timestamp].
    - Row 2: Alice (London) | `is_current` = **true** | `version_number` = **2**.
    - Row 3: Bob (Chicago) | `is_current` = true.

### State 3: Verifying the Audit Trail

- **Action:** Show the `history()` output.
- **Result:** Point out the `WRITE` and `MERGE` operations. This is the "secret sauce" of Delta Lake—the **Transaction Log**.

### State 4: The Rollback

- **Action:** Run the `audit_rollback` or `restoreToVersion` command.
- **Result:** Show how Alice is back in New York as the "Current" record, but the "London" record is still visible in the history (V3).

---

## 4. Key Takeaways for your Audience

- **Data Integrity:** SCD Type 2 ensures we never lose a piece of information, even if it changes.
- **Scalability:** Delta Lake handles this on millions of rows across a distributed cluster, whereas traditional PostgreSQL might struggle with very large datasets.
- **Compliance:** This implementation satisfies **GDPR/Audit** requirements because it clearly shows *who* changed *what* and *when*.
-

Below is a step-by-step flow you can follow to complete that hands-on end to end (PostgreSQL RDS SCD2 + stored procedures + Delta Lake SCD + Git/version approvals/rollback/audit).

## 1) Define the target entity (what you SCD2)

Pick one master entity from your NYC taxi work (example: Taxi Zone master).

- Business key: `zone_id` (or `locationid`)
- Tracked attributes: `zone_name`, `borough`, `service_zone`, etc.
- Source "incoming" table: `stg_zone_updates` (new snapshot / incremental changes)

Goal: keep full history of attribute changes.

## 2) Create PostgreSQL schemas & tables (RDS)

### 2.1 Create a staging table (incoming data)

Use staging for the daily/weekly snapshot you ingest.

```sql
CREATE SCHEMA IF NOT EXISTS stg;
CREATE SCHEMA IF NOT EXISTS mdm;
CREATE SCHEMA IF NOT EXISTS gov;
sql
CREATE TABLE IF NOT EXISTS stg.zone_updates (
  zone_id       INT NOT NULL,
  zone_name     TEXT,
  borough       TEXT,
  service_zone  TEXT,
  source_file   TEXT,
  ingested_at   TIMESTAMP DEFAULT now()
);
```

## 2.2 Create the SCD Type 2 dimension with metadata columns

Key columns for SCD2:

- `effective_from`, `effective_to`
- `is_current`
- `version_num`
- governance metadata: `created_by`, `approved_by`, `approval_reason`, etc.

```
sql
CREATE TABLE IF NOT EXISTS mdm.dim_zone_scd2 (
  dim_zone_sk     BIGSERIAL PRIMARY KEY,
  zone_id         INT NOT NULL,              -- business key
  zone_name       TEXT,
  borough         TEXT,
  service_zone    TEXT,

  effective_from  TIMESTAMP NOT NULL,
  effective_to    TIMESTAMP NOT NULL DEFAULT '9999-12-31',
  is_current      BOOLEAN NOT NULL DEFAULT true,
  version_num     INT NOT NULL,

  record_hash     TEXT,                      -- helps detect changes
  source_system   TEXT DEFAULT 'nyc_taxi',
  created_at      TIMESTAMP NOT NULL DEFAULT now(),
  created_by      TEXT DEFAULT current_user,

  approval_status TEXT NOT NULL DEFAULT 'PENDING',  -- PENDING/APPROVED/REJECTED
  approved_at     TIMESTAMP,
  approved_by     TEXT,
  approval_reason TEXT,

  updated_at      TIMESTAMP NOT NULL DEFAULT now()
);

CREATE INDEX IF NOT EXISTS idx_dim_zone_current
ON mdm.dim_zone_scd2(zone_id)
WHERE is_current = true;
```

## 3) Write stored procedures for SCD2 operations

### 3.1 Helper: compute a “change hash”

In practice you can hash the tracked attributes. In PostgreSQL:

```sql
-- optional extension
CREATE EXTENSION IF NOT EXISTS pgcrypto;
```

Hash expression example (used inside procedure):

```
encode(digest(coalesce(zone_name,'') || '|' || coalesce(borough,'') ||
'|' || coalesce(service_zone,''), 'sha256'),'hex')
```

### 3.2 Procedure: upsert SCD2 from staging

Flow:

1. For each incoming row, find current row in `mdm.dim_zone_scd2`.
2. If not exists → insert version 1.
3. If exists but hash differs → “close” current row and insert new version.

```sql
CREATE OR REPLACE PROCEDURE mdm.apply_zone_scd2(p_run_by TEXT DEFAULT current_user)
LANGUAGE plpgsql
AS $$
DECLARE
 r RECORD;
 v_hash TEXT;
 v_curr RECORD;
 v_next_version INT;
BEGIN
 FOR r IN SELECT * FROM stg.zone_updates LOOP

   v_hash := encode(digest(
     coalesce(r.zone_name,'') || '|' || coalesce(r.borough,'') || '|' || coalesce(r.service_zone,''),
     'sha256'
   ), 'hex');

   SELECT * INTO v_curr
   FROM mdm.dim_zone_scd2
   WHERE zone_id = r.zone_id AND is_current = true
   LIMIT 1;

   IF NOT FOUND THEN
     INSERT INTO mdm.dim_zone_scd2 (
       zone_id, zone_name, borough, service_zone,
       effective_from, effective_to, is_current,
       version_num, record_hash, created_by
     )
     VALUES (
       r.zone_id, r.zone_name, r.borough, r.service_zone,
       now(), '9999-12-31', true,
       1, v_hash, p_run_by
     );
   ELSE
     IF v_curr.record_hash <> v_hash THEN
       v_next_version := v_curr.version_num + 1;

       UPDATE mdm.dim_zone_scd2
       SET effective_to = now(),
           is_current = false,
           updated_at = now()
       WHERE dim_zone_sk = v_curr.dim_zone_sk;

       INSERT INTO mdm.dim_zone_scd2 (
         zone_id, zone_name, borough, service_zone,
         effective_from, effective_to, is_current,
         version_num, record_hash, created_by
       )
       VALUES (
         r.zone_id, r.zone_name, r.borough, r.service_zone,
         now(), '9999-12-31', true,
         v_next_version, v_hash, p_run_by
       );
     END IF;
   END IF;

 END LOOP;
END $$;
```

Run:

```sql
CALL mdm.apply_zone_scd2('pipeline_job');
```

## 4) Create version management procedures (approve/rollback/audit)

### 4.1 approve_version(record_id, approver, reason)

Here `record_id` can be the surrogate key (`dim_zone_sk`) or `(zone_id, version_num)`. Use `dim_zone_sk` for simplicity.

sql

```
CREATE OR REPLACE PROCEDURE gov.approve_version(p_dim_zone_sk BIGINT, p_approver TEXT,
p_reason TEXT)
LANGUAGE plpgsql
AS $$
BEGIN
 UPDATE mdm.dim_zone_scd2
 SET approval_status = 'APPROVED',
     approved_by = p_approver,
     approved_at = now(),
     approval_reason = p_reason,
     updated_at = now()
 WHERE dim_zone_sk = p_dim_zone_sk;
END $$;
```

## 4.2 rollback_version(record_id, target_version, reason)

Rollback means: for a given `zone_id`, make `target_version` the current one again.

sql
```
CREATE OR REPLACE PROCEDURE gov.rollback_version(p_zone_id INT, p_target_version INT, p_reason
TEXT)
LANGUAGE plpgsql
AS $$
BEGIN
 -- Close current row
 UPDATE mdm.dim_zone_scd2
 SET is_current = false,
     effective_to = now(),
     updated_at = now()
 WHERE zone_id = p_zone_id AND is_current = true;

 -- Re-open the target version (make it current again)
 UPDATE mdm.dim_zone_scd2
 SET is_current = true,
     effective_to = '9999-12-31',
     updated_at = now(),
     approval_status = 'APPROVED',
     approval_reason = coalesce(approval_reason,'') || ' | ROLLBACK: ' || p_reason
 WHERE zone_id = p_zone_id AND version_num = p_target_version;
END $$;
```

## 4.3 audit_version_history(record_id, date_range)

Use a function returning rows (easier than procedure for “select”).

sql
```
CREATE OR REPLACE FUNCTION gov.audit_version_history(p_zone_id INT, p_from TIMESTAMP, p_to
TIMESTAMP)
RETURNS TABLE (
 zone_id INT,
 version_num INT,
 effective_from TIMESTAMP,
 effective_to TIMESTAMP,
 is_current BOOLEAN,
 approval_status TEXT,
 approved_by TEXT,
 approved_at TIMESTAMP,
 approval_reason TEXT,
 created_at TIMESTAMP,
 created_by TEXT
)
LANGUAGE sql
AS $$
 SELECT
   zone_id, version_num, effective_from, effective_to, is_current,
   approval_status, approved_by, approved_at, approval_reason,
   created_at, created_by
 FROM mdm.dim_zone_scd2
 WHERE zone_id = p_zone_id
   AND effective_from <= p_to
   AND effective_to >= p_from
 ORDER BY version_num;
$$;
```

Example:

```sql
SELECT * FROM gov.audit_version_history(132, '2026-01-01', '2026-12-31');
```

## 5) PySpark + Delta Lake SCD2 with time travel rollback

This is the “lake” version of the same idea, stored as a Delta table.

### 5.1 Create Delta table path

Example S3 path:

- `s3://<bucket>/delta/mdm/dim_zone_scd2/`

### 5.2 PySpark script outline (SCD2 merge)

High-level flow:

1. Read source updates (CSV/Parquet) into `df_updates`.
2. Read Delta target `df_dim`.
3. Identify changed/new rows using hash.

4. MERGE:
   - Close current version
     (set `is_current=false`, `effective_to=current_timestamp`)
   - Insert new version with incremented version_num

Pseudo-code (common pattern):

```
python
from pyspark.sql import functions as F, Window
from delta.tables import DeltaTable

target_path = "s3://<bucket>/delta/mdm/dim_zone_scd2"
updates_path = "s3://<bucket>/stg/zone_updates/"

updates = (spark.read.format("parquet").load(updates_path)
 .withColumn("record_hash", F.sha2(F.concat_ws("|",
    F.coalesce(F.col("zone_name"), F.lit("")),
    F.coalesce(F.col("borough"), F.lit("")),
    F.coalesce(F.col("service_zone"), F.lit(""))
 ), 256))
)

dim = DeltaTable.forPath(spark, target_path)

# merge keys
cond = "t.zone_id = s.zone_id AND t.is_current = true"

(dim.alias("t")
 .merge(updates.alias("s"), cond)
 .whenMatchedUpdate(
   condition="t.record_hash <> s.record_hash",
   set={
    "is_current": F.lit(False),
    "effective_to": F.current_timestamp(),
    "updated_at": F.current_timestamp()
   }
 )
 .whenNotMatchedInsert(values={
   "zone_id": F.col("s.zone_id"),
   "zone_name": F.col("s.zone_name"),
   "borough": F.col("s.borough"),
   "service_zone": F.col("s.service_zone"),
   "effective_from": F.current_timestamp(),
   "effective_to": F.lit("9999-12-31"),
   "is_current": F.lit(True),
   "version_num": F.lit(1),  # (for real impl, compute next version)
   "record_hash": F.col("s.record_hash"),
   "created_at": F.current_timestamp(),
   "updated_at": F.current_timestamp()
 })
 .execute()
```

)

Note: incrementing `version_num` properly usually requires:

- computing “next version per zone_id” from the target table, then joining to updates before insert.

### 5.3 Delta Lake rollback using time travel

Flow:

1. Find previous table version.
2. Restore table to that version.

Example queries:

```sql
DESCRIBE HISTORY delta.`s3://<bucket>/delta/mdm/dim_zone_scd2`;
```

Then rollback:

```sql
RESTORE TABLE delta.`s3://<bucket>/delta/mdm/dim_zone_scd2` TO VERSION AS OF 25;
```

(Or restore to timestamp.)

## 6) Implement in SQL with version control (Git flow)

Recommended repo layout:

```text
sql/
  rds/
    001_create_tables.sql
    002_apply_scd2_proc.sql
    003_gov_procs.sql
  delta/
    001_dim_zone_scd2.sql
spark/
  scd2_dim_zone.py
docs/
  scd2_design.md
```

Flow:

1. One change = one branch.
2. Every SQL file has a governance header (Owner, Classification, Change history).
3. PR requires review; once merged, deploy (run SQL files against RDS / run Spark job).

## Suggested execution order (end-to-end flow)

1. Load zone updates into `stg.zone_updates` (or via pipeline).
2. Run `CALL mdm.apply_zone_scd2('pipeline_job');`
3. Approve new versions (optional governance step): `CALL gov.approve_version(<dim_zone_sk>, 'alice', 'validated source');`
4. Audit: `SELECT * FROM gov.audit_version_history(zone_id, from_ts, to_ts);`
5. If rollback required: `CALL gov.rollback_version(zone_id, target_version, 'bad input file');`
6. Run Delta SCD job in Spark (lake version) and test time travel rollback with `DESCRIBE HISTORY` + `RESTORE`.

If you share your exact entity (zone/vendor/customer) and the columns you have in RDS today, the SQL/procedures can be adjusted to your real table and column names.