DATA MODELLING 101 (2)

Slowly Changing Dimensions (SCDs)

Making Change a First-Class Citizen in Your Data Model

Based on Ralph Kimball's Data Warehouse Toolkit

Milan Gabriel
Senior Data Engineer

Dimension Tables Aren't Actually Static

"What happens to your fact table's history when the dimension it points to changes?"

Real-world entities change over time, and your model must account for it to prevent data corruption.

- Session 1 built the star schema - facts measure, dimensions describe.
- The gap: We treated dimensions as fixed. In reality, customers move, products change categories, and employees change departments.

THE ANALYTICAL RISK

Overwriting dimension rows causes **"silent data corruption"**- past sales get re-attributed to new attributes without any error or crash.

The Naive UPDATE Breaks Historical Reporting

Updating in place destroys the integrity of historical performance reports.

JANUARY

Initial State

Rahul lives in **Mumbai**.

Purchases ₹50,000 worth of goods.

✓ **Correct Attribution**

JUNE

The Change

Rahul relocates to **Bangalore**.

Database is UPDATEd in place:

```
SET city = 'Bangalore'
```

RESULT

The Corruption

Run "Sales by Region - Jan" today.

Rahul's Jan sale now shows as **Bangalore**.

✗ **Retroactive Error**

THE BOTTOM LINE

Regional performance reports become retroactively wrong. SCDs exist to decide **deliberately** what happens to history when attributes change.

Four Ways to Handle Change

The choice of SCD type is a business decision, not a technical default.

Type 0: Retain Original

Locked at first load. Never changes, preserving the initial state forever.

Type 1: Overwrite

Update in place. No history is kept. Simple but destroys past context.

Type 2: Add New Row

Full history preserved. A new row is added for every change.

Type 3: Add New Column

Limited history. One extra column stores the previous value.

THE DECISION: Four deliberate choices - not four difficulty levels.

Type 0 - The Value That Never Changes

Use Type 0 for immutable attributes that define the start of a relationship.

The attribute is fixed at the moment the record is first created and is never updated again, even if the source system changes it.

Common Use Cases

- Original Signup Date
- Date of Birth
- First-Touch Acquisition Channel
- Credit Score at Loan Approval

THE GOLDEN RULE

If your source system changes it, your warehouse should **NOT**. Some business questions only make sense with the original value.

SCD Type 1 – Update in Place, Lose the Past

Type 1 is for corrections and non-analytical attributes where history is irrelevant.

WHEN TO USE IT

- **Corrections:** Fixing a typo in a customer's name or address.
- **Non-Analytical:** Updating a wrong phone number or email address.
- **Low Value:** Attributes where history genuinely doesn't matter for trend analysis.

THE TRADE-OFF

PROS	CONS
✓ Simple to implement	✗ Destroys historical context
✓ Maintains small table size	✗ Silently re-maps past facts
✓ Fast query performance	✗ No audit trail of changes

THE GOLDEN RULE

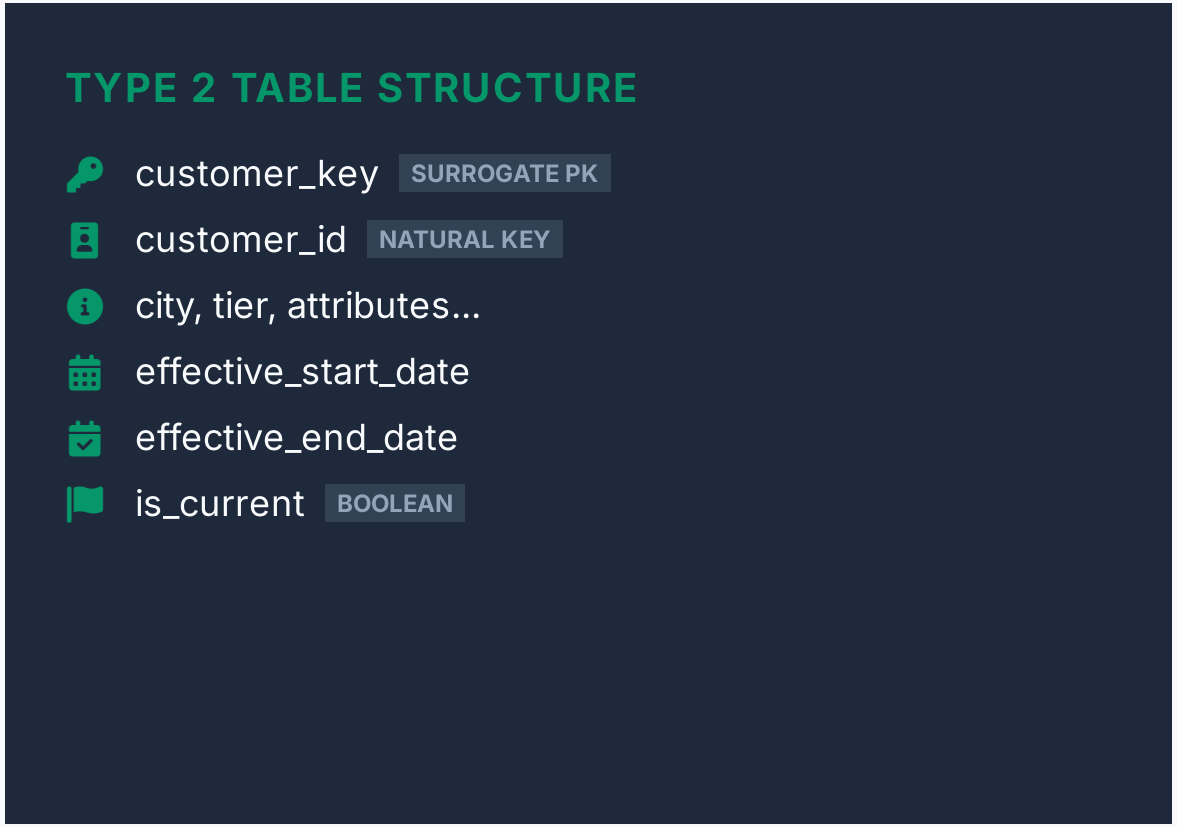
Never use Type 1 on an attribute your fact table analysis depends on for trend history (e.g., Region, Category, Price Tier).

SCD Type 2 – Preserve Every Version as History

Type 2 is the gold standard for historical tracking in dimensional modeling.

When an attribute changes, don't touch the old row. Instead, insert a new row with the new value and mark which row is "current."

Why it Matters: Facts join to the version that was current at the time of the transaction, freezing history exactly as it happened.



> **Surrogate Key ≠ Natural Key**
>
> The natural key ties versions together; the surrogate key makes each version a unique, joinable row for the fact table.

Type 3 - Track the Previous Value

Lightweight tracking for a single transition without row growth.

When an attribute changes, keep the current value in its column and push the old value into a new "previous" column. One row, no growth.

THE CONSTRAINT

Only remembers **one step back**. A second change overwrites the "previous" value - the first-ever value is gone.

Table Structure

- customer_key (PK stays same)
- customer_id
- current_region
- previous_region
- region_change_date

Best Use Case

- **Sales Territory Realignment:** Leadership wants "old region vs. new region" side by side for a one-time transition.
- **Department Reorgs:** Comparing metrics before and after a single organizational shift.

Comparing SCD Types 0, 1, 2, and 3

Choose the type based on whether you need a full timeline or just a snapshot.

Feature	Type 0: Retain	Type 1: Overwrite	Type 2: Add Row	Type 3: New Col
History Kept	None (Frozen)	**None (Overwritten)**	**Full History**	One step back
Row Count	Static	Static	Grows over time	Static
Update Method	Never updated	UPDATE in place	INSERT new row	UPDATE in place
Key Used	Natural Key	Natural Key	**Surrogate Key**	Natural Key
Use Case	Immutable facts	Corrections	Trend attributes	One-time compare
Fact Table Join	Unaffected	**Silently re-mapped**	**Point-in-time**	Unaffected

Key Insight: Type 2 gives you a timeline. Type 3 gives you a snapshot of one transition. Don't use Type 3 where you'll need to ask 'what was it two changes ago.'

Before You Can Apply a Change, You Must Detect One

Hash-based change detection is the most efficient way to track multi-column updates.

THE HASHING STRATEGY

- **Efficiency:** Comparing a single hash value is significantly faster than checking 15+ columns one-by-one.
- **Reliability:** Reduces the risk of missing a change in a specific column due to manual SQL errors.
- **Selective Tracking:** Only hash the business attributes you care about-exclude audit columns and keys.

```sql
SELECT s.*
FROM staging s
JOIN dim_customer d
  ON s.customer_id = d.customer_id
  AND d.is_current = TRUE
WHERE s.row_hash <> d.row_hash -- Change
detected
```

Pro Tip: Use MD5 or SHA-256 to generate the `row_hash` in your staging layer. This makes your ETL logic cleaner and more maintainable.

Applying a Type 2 Change is Always Two Statements

Atomic transactions are non-negotiable to prevent broken joins and data gaps.

Step 1: Close the Old Row

Update the existing "current" record to mark it as historical. Set the end date to today.

Step 2: Insert the New Version

Add a fresh row with the updated attributes, a new surrogate key, and an open-ended end date.

⚠ CRITICAL WARNING

These must run in the **same transaction**. Failure in Step 2 after Step 1 succeeds leaves a customer with zero current rows, breaking every future join.

STEP 1: CLOSE OLD ROW

```
UPDATE dim_customer
SET effective_end_date = CURRENT_DATE,
    is_current = FALSE
WHERE customer_id = :changed_id
  AND is_current = TRUE;
```

STEP 2: INSERT NEW VERSION

```
INSERT INTO dim_customer
(customer_id, city, tier, effective_start_date,
 effective_end_date, is_current)
VALUES
(:changed_id, :new_city, :new_tier, CURRENT_DATE,
 '9999-12-31', TRUE);
```

In PySpark, MERGE Replaces Both Statements at Once

Delta Lake's MERGE INTO provides an atomic, high-performance way to handle SCD Type 2.

THE DELTA ADVANTAGE

Delta Lake's `MERGE INTO` lets you express "close old row + insert new row" as a single atomic operation.

- **Atomicity:** No partial-failure risk between update and insert.
- **Performance:** Optimized for distributed big data workloads.
- **Declarative:** Express logic clearly without manual transaction management.

> *Note: This closes the changed row in one pass. New-version rows still need a second insert-only pass (or a union staging trick) since one MERGE can't close AND insert the same logical customer in a single matched branch.*

```
from delta.tables import DeltaTable

target = DeltaTable.forName(spark, "dim_customer")

(target.alias("t")
  .merge(staging_df.alias("s"), "t.id = s.id AND t.is_current = true")
  .whenMatchedUpdate(
    condition="t.row_hash <> s.row_hash",
    set={
      "is_current": "false",
      "effective_end_date": current_date()
    }
  )
  .whenNotMatchedInsert(values={
    "customer_id": "s.customer_id",
    "city": "s.city",
    "row_hash": "s.row_hash",
    "effective_start_date": current_date(),
    "effective_end_date": '9999-12-31',
    "is_current": "true"
  })
  .execute()
)
```

Avoid These Five Common SCD Mistakes

Most failures stem from implementation shortcuts taken under pressure.

Natural Key as PK

Breaks Type 2 immediately. Entities need multiple rows with unique keys. Always use Surrogate Keys.

Forgetting to Close Rows

Leaves two "current" rows for one customer, causing fan-out joins and duplicated facts.

Hashing Audit Columns

Including load_timestamp triggers false positives on every load, even with no business change.

Type 2 on Everything

Not every column needs history. Over-tracking bloats the dimension and slows every query.

Using Type 3 for Ongoing Change

Fine for a one-time reorg. Breaks down fast if the attribute keeps changing - you lose everything before the "previous" value.

Five Rules to Master SCDs

Consistency and technical rigor prevent the most expensive data warehouse bugs.

Detect Before You Apply

01

Never write a change you haven't first confirmed with a hash or column comparison. Avoid redundant updates.

One Transaction, Two Statements

02

Close and insert together, or not at all. Atomic operations are the only way to guarantee data integrity.

Surrogate Keys are Non-Negotiable

03

The natural key identifies the entity; the surrogate key identifies the version. You cannot track Type 2 without them.

Match the Type to the Business

04

One hop of history? Type 3. Full timeline? Type 2. Choose the method that matches the analytical requirement.

MERGE is a Tool, Not a Shortcut

05

It removes race conditions, but you still need to design the close/insert logic correctly for your specific engine.

THE RESULT: These rules prevent duplicated current rows and silently rewritten history.

Advance Your Skills with Kimball's Methodology

Real-world modeling requires continuous practice and learning.

ESSENTIAL READING & PRACTICE

The Data Warehouse Toolkit
Ralph Kimball (Chapter 5 - Slowly Changing Dimensions)

Immediate Practice:
Redesign a Session 1 dimension as Type 2 and write the hash-based change-detection query.

NEXT TOPICS & CONTACT

Coming Up:
Fact Table Pattern Selection,Grain Changes, Conformed Dimensions & Bus Architecture.

Get in Touch:
linkedin.com/in/milan-gabriel
bit.ly/the1onwrongway

Go forth and never lose a row of history.