

🌟 Status	Done
📅 Due	@November 21, 2024
📍 Project	Data Expert.io
☰ Sub-tasks	🟡 day2 lab - scd
🏷️ Tags	

Dimensional data modeling [week1day2]



Week 1 day 2: This week we are focusing all our attention on dimension data modeling especially understanding Slowly changing Dimensions and building idempotent pipelines for robust data systems.

Key concepts this week

1. Idempotent pipelines
2. Slowly changing dimensions

Idempotent pipelines

Idempotent pipelines are pipelines that produce the same results (either in backfill or production) regardless of date/time or how many times you run it.

Example:

Imagine a data pipeline that processes transactions and writes them to a database:

1. **Non-Idempotent Pipeline:**

If you re-run the pipeline, the same transactions may be written multiple times, causing duplicates in the database.

2. Idempotent Pipeline:

The pipeline checks if a transaction already exists before writing it. This way, re-running the pipeline doesn't change the result—only new transactions are added, and duplicates are avoided.

Why Troubleshooting Non-Idempotent Pipelines is Hard

1. Backfilling leads to inconsistencies in old and restored data
2. Silent Failures: Non-idempotent pipelines can fail quietly without immediate signs of error. For example, duplicate records might be added or data might be overwritten. These issues are often discovered when a **data analyst** or end user notices data inconsistencies.
3. Unit testing cannot replicate the production behavior
4. Hard to debug

Cause and fix

what causes the pipelines to become non-idempotent and how to fix it

1. **Using `INSERT INTO` Without `TRUNCATE`**: If you keep appending rows to a table without clearing old data, duplicates will accumulate when the pipeline is re-run.
 - **Fix**: Use `MERGE` or `INSERT OVERWRITE` to ensure the table is updated without creating duplicates.

```
INSERT OVERWRITE TABLE sales
SELECT date, sale_id, amount
FROM daily_sales_file;

-- OR --

MERGE INTO sales AS target
USING daily_sales_file AS source
ON target.sale_id = source.sale_id
   AND target.date = source.date
WHEN MATCHED THEN
    UPDATE SET amount = source.amount
WHEN NOT MATCHED THEN
    INSERT (date, sale_id, amount)
    VALUES (source.date, source.sale_id, source.amount);
```

2. **Using a `start_date` Without a Corresponding `end_date`** : Filtering data with only a `start_date` can lead to overlapping or missing ranges if the pipeline runs multiple times.
 - **Fix:** Always use both `start_date` and `end_date` to clearly define the range of data being processed.
3. **Not Using a Full Set of Partition Sensors:** If your pipeline processes data based on partitions (e.g., dates) but does not validate that all required partitions exist, partial or incomplete data might be processed.
 - **Fix:** Ensure all expected partitions are available before running the pipeline
4. **Not Using `depends_on_past` for Cumulative Pipelines:**
 - For pipelines that build incrementally, skipping or ignoring dependencies on previous runs can lead to gaps or duplicated data.
 - **Fix:** Use `depends_on_past` to ensure each step relies on the successful completion of the previous step.
 - mostly in airflow

```
task = PythonOperator(  
    task_id='process_weekly_totals',  
    python_callable=process_weekly_totals,  
    depends_on_past=True, # Ensure it only runs after the previous week's task is successful  
)
```

The Problem with Relying on the "Latest" Partition

Slowly Changing Dimension tables track how things change over time, like user statuses or product prices. But when a pipeline processes only the "latest" partition, it can miss critical updates, leading to incomplete or inconsistent data.

Imagine Facebook tracking whether accounts are "fake" or "not fake." An account's status changes over time:

1. Nov 1, 2024: The account is flagged as "fake."
2. Nov 15, 2024: After an appeal, it's marked "not fake."
3. Nov 20, 2024: Suspicious activity gets it flagged as "fake" again.

If the pipeline only looks at the "latest" partition (Nov 20), it treats the account as "fake" without considering the earlier changes. This causes two major issues:

1. **Broken Reporting:** Facebook misses the Nov 1 flag and underreports how many accounts were flagged as "fake" over time.
2. **Policy Failures:** Rules for suspending accounts flagged multiple times don't trigger because the earlier "fake" status is ignored.

A Better Approach

1. **Keep the Full History:** Record each status change with start and end dates to track the complete lifecycle.

```

account_id | status      | start_date | end_date
-----|-----|-----|-----
123        | fake        | 2024-11-01 | 2024-11-15
123        | not_fake    | 2024-11-15 | 2024-11-20
123        | fake        | 2024-11-20 | NULL

```




2. **Avoid Overwrites:** Add new rows for changes instead of replacing old data.
3. **Process Everything:** Analyze all historical partitions to ensure no updates are missed.


By capturing the complete story, Facebook can report accurately, enforce rules effectively, and avoid data inconsistencies.

Slowly Changing Dimensions

SCD is a way to track changes in your data over time. Instead of just storing the most recent version of data, SCDs allow you to maintain a history of all changes.

SCD Types:

Type	SCD Features	Idempotent?	Reason
0	Cannot be labeled as SCD (value has zero change, e.g., birth date).	Yes 	The value is always the same.
1	Tracks only the latest value of a dimension.	No 	When backfilling, it shows the current state, not the historical state.
2	Tracks full history with <code>START_DATE</code> and <code>END_DATE</code> .	Yes 	Captures a window of time, ensuring all changes are

			tracked.
3	Holds two values: "original" and "current."	No 	Backfilling cannot determine when to reference the original or current value.

1. Type 0: Cannot be labeled as SCD (value has zero change, e.g., birth date).
2. Type 1 (No History):
 - Simply overwrite the old data with the new data.
 - Example: If a customer's address changes, you just replace the old address with the new one.
 - Use case: When history is not important.
3. Type 2 (Full History):
 - Maintain a new record for each change, with start and end dates for each version.

```
customer_id | address          | start_date | end_date
-----|-----|-----|-----
1          | Old Address     | 2024-01-01 | 2024-06-30
1          | New Address     | 2024-07-01 | NULL
```

- Use case: When you need to track changes over time.
4. Type 3 (Limited History):
 - Store some history in additional columns (e.g., current value and previous value).

```
customer_id | current_address | previous_address
-----|-----|-----
1          | New Address     | Old Address
```

- Use case: When you only need to track recent changes.

Idempotent: Use **Type 0** or **Type 2** for consistent, reliable pipelines.

Not Idempotent: Avoid **Type 1** and **Type 3** if history matters.

Why SCDs are Controversial

Max (creator of Airflow) is critical of SCDs because they can be complex and prone to errors, particularly in distributed systems like Airflow. Maintaining and querying SCDs can lead to bugs and inconsistencies.

Alternatives?

1. Latest Snapshot: stores only the most recent state of your data. It is straightforward to implement, requires minimal storage, and supports fast queries, making it an efficient choice for real-time reporting or applications where historical data isn't needed. However, its simplicity comes at a cost—there is no way to track historical changes, which limits its usefulness for long-term analysis, trend tracking, or audits.
 - Only store the most recent state of your data.
 - Pros: Simple, minimal storage, fast queries.
 - Cons: No historical tracking.
2. Daily/Monthly/Yearly Snapshots: capture a complete copy of your data at regular intervals, such as daily, monthly, or yearly. This method simplifies historical tracking since each snapshot represents a point-in-time view of your data. It eliminates the need to track individual changes and is useful for identifying broader trends over time. However, this approach is storage-intensive and can make pinpointing the exact timestamps of changes challenging, especially when changes occur between snapshot intervals.
 - Store a complete snapshot of your data periodically.
 - Pros: Easier to track history than SCDs, no need to track individual changes.
 - Cons: High storage cost, harder to pinpoint exact change timestamps.

How to Decide?

1. How Slowly Do Dimensions Change?

- For rarely changing data, SCDs may be unnecessary.
- For frequently updated critical data (e.g., user status), SCDs are more appropriate.

2. Business Needs:

- Use a **latest snapshot** if only the current state matters (e.g., a live dashboard showing product stock).
- Use **periodic snapshots** for basic history without granular changes (e.g., monthly sales reports).
- Use **SCDs** for detailed tracking and analysis (e.g., monitoring changes for regulatory compliance).