# Technical Documentation

## Problem Statement

The data engineering team aims to create a solution that allows the data science team to save datasets in the data warehouse with a maintained history of data versions. The solution should meet the following requirements :-

- User-Friendly and abstracted Access
- Access to Current Data Version
- Access to Historical Data Versions
- List of All Versions
- Schema and Data Versioning
- Seamless Data Warehouse Migration
- Additional Features (Optional): Add 1-3 extra features that will add value for end users

## Solution Overview

How important is dataset versioning? Dataset versioning in a data warehouse boosts data accuracy and traceability by keeping a clear history of changes and enabling rollbacks when needed. This ensures data integrity, supports audits, and makes it easier for teams to collaborate and experiment without risking production data, ultimately leading to better decision-making.

The first step in integrating dataset versioning features into a data warehouse is knowing in which layer changes should be tracked. Keeping track of dataset versions starts at the modeling layer because at this stage, data transformations are complete and structured datasets are prepared for analytics and modeling. Versioning at the modeling layer also allows us to track changes such as schema changes, and row-level changes.

The presentation layer can also leverage data versioning for providing users with access to the different versions. This is where you create views, stored procedures, and user-friendly interfaces that abstract the versioning logic and make it accessible.

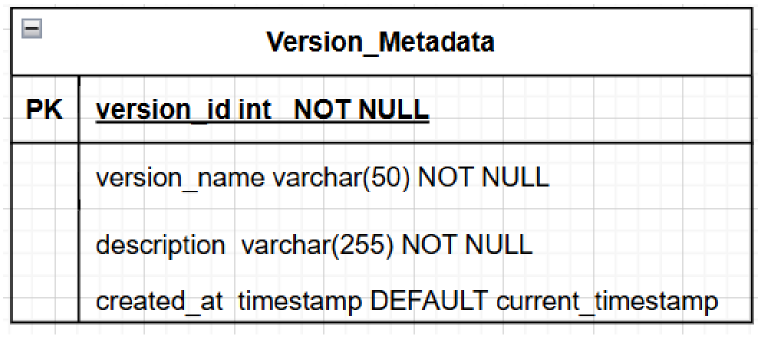
### Versioning Approach:

- **Modeling Layer**: Store historical versions within version-controlled tables or partitions in the data warehouse. Implement row and column versioning to handle data and schema changes.
- **Presentation Layer**: Provide access interfaces, such as views, that users can interact with to retrieve specific versions or metadata about the versions.

**In summary**, the versioning solution should be built at the **modeling layer**, where datasets are structured and prepared for use, and should be **exposed in the presentation layer** to allow users easy access to historical and current versions. This setup ensures consistency and availability for data science needs without impacting the underlying source or staging data.

**Step # 1**

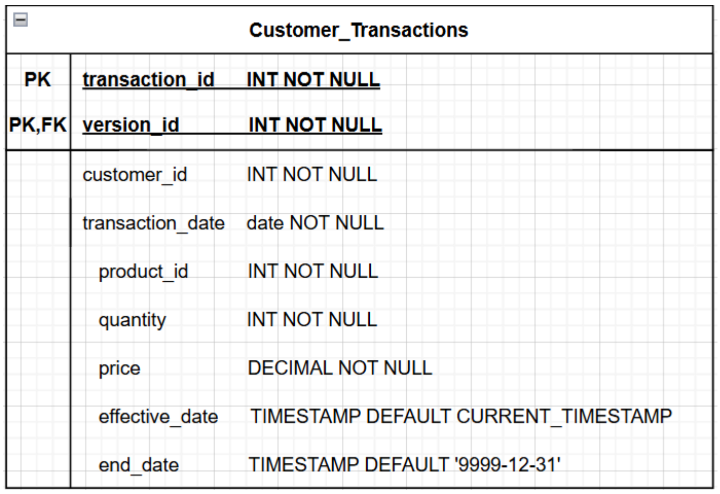
The first step in creating this solution is to have a version metadata tale to track versions, when each version was created and its description.



| Version_Metadata | |
|---|---|
| PK | version_id int NOT NULL |
| | version_name varchar(50) NOT NULL |
| | description varchar(255) NOT NULL |
| | created_at timestamp DEFAULT current_timestamp |

**Step # 2**

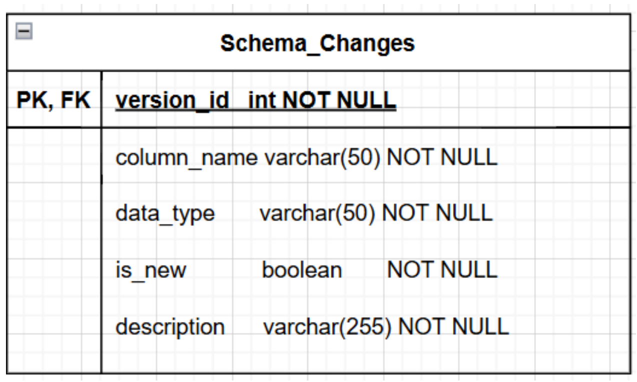
The second step is that in the modeling layer, when the transformations are done, we create the table with some extra columns to include row-level versioning. This will be implemented for each new dataset that comes through the pipeline. This is shown in a sample customer_transactions table below. The effecitve_date field is when this row was added and end_date changes when a row is updated.



| Customer_Transactions | | |
|---|---|---|
| PK | transaction_id | INT NOT NULL |
| PK,FK | version_id | INT NOT NULL |
| | customer_id | INT NOT NULL |
| | transaction_date | date NOT NULL |
| | product_id | INT NOT NULL |
| | quantity | INT NOT NULL |
| | price | DECIMAL NOT NULL |
| | effective_date | TIMESTAMP DEFAULT CURRENT_TIMESTAMP |
| | end_date | TIMESTAMP DEFAULT '9999-12-31' |

**Step # 3**

In this step, we create a table to track schema changes which include the field name, data type, flag for checking if it's a new or old field and its description. Having the description of each field is useful for understanding the data definitions.



**Schema_Changes**

| PK, FK | **version_id int NOT NULL** |
|---|---|
| | column_name varchar(50) NOT NULL |
| | data_type varchar(50) NOT NULL |
| | is_new boolean NOT NULL |
| | description varchar(255) NOT NULL |

**Presentation Layer**

In the presentation layer, which is usually an interface to access the data and run queries, I choose to create a few stored procedures to make it easy for users to interact with the data warehouse without the need to write their own code.

These are the Stored Procedures that can be called by users:

Check sample SQL code here

- Retrieve latest versions
- Retrieve specific version
- List all available versions

**Example Workflow:**

- **Initial Data Load**: Populate `version_metadata` with `version_id = 1` and load initial `customer_transactions`.
- **Monthly Updates**: For each new update (e.g., November data), create `version_id = 2`, add new data with this version ID, and update `schema_changes` if columns are added or modified.
- **User Access**: Data scientists query `current_customer_transactions` for the latest data or use the version ID to access historical data as needed.

This structure ensures data and schema changes are versioned, accessible, and easy to query, while the abstraction in the presentation layer keeps the user experience simple.

## Performance & Storage

For enhancing *performance* :-

- Use indexing on key columns like `version_id`, `effective_date`, and `end_date` in the `customer_transactions` table. This minimizes full table scans.

```
CREATE INDEX idx_version_id ON customer_transactions(version_id);
CREATE INDEX idx_effective_date ON customer_transactions(effective_date);
```

- Design the ETL/ELT pipeline to append only the changed rows (new, modified, or deleted) to the `customer_transactions` table rather than replacing the entire dataset. This approach ensures that only new or updated data is processed and stored, which reduces compute time and resource usage.
- Partition the `customer_transactions` table by a key such as `version_id` or `effective_date`. Partitioning breaks down the table into smaller, more manageable pieces, so queries only scan relevant partitions, improving performance.

For enhancing *storage*:-

- Use versioning to store only the records that have changed (new, updated, or deleted). Older versions are stored with an `end_date` while the current version remains active with an `effective_date`. By storing only the changes and marking previous versions with `end_date`, this avoids duplicating entire datasets.
- The same approach is used to track schema changes.
- Archive older versions of data that are infrequently accessed by moving them to cheaper storage tiers (e.g., cold storage) or delete if no longer needed.
- Set up automatic processes for managing the lifecycle of the data, including archiving, purging old data, and compressing versions that haven't been queried in a while.

For enhancing *migration*:-

The data versioning solution helps in database migration by:

1. Minimizing Downtime: Incremental migration of partitions ensures minimal disruption, allowing users to access the latest data while older versions are being transferred.
2. Simplified Data Validation: Post-migration, you can easily compare the new and old versions to confirm data accuracy and integrity.
3. Flexibility with Schema Changes: The solution can adapt to new database systems, allowing schema changes to be tracked and applied.
4. Efficient Large Dataset Migration: Partitioning helps break large datasets into smaller chunks, making the migration process faster and more manageable.
5. Rollback Capability: Historical versions ensure that you can roll back to a previous state if issues arise during migration.

## Scaling the Dataset Versioning Solution to Multiple Tables

The versioning solution described so far is based on a single table (`customer_transactions`). As the data warehouse grows, we need to extend this approach to multiple tables. Here's how we can scale the solution:

1. **Centralized Versioning Metadata**:
   We extend the `version_metadata` table to track versioning details for each table, including version IDs, effective dates, and schema versions. If the number of versions per table increases drastically, we can create a partition on table name so that all versions of a table are stored in their own partition for easier querying.
2. **Schema Changes Management**:
   The `schema_changes` table tracks schema changes for each table, providing a clear history of schema evolution. So each new dataset has its own schema changes table
3. **Storage Management**:
   Each table can have its own archiving and purging policies. Older versions are archived to cheaper storage, and unnecessary data is purged or compressed as needed.
4. **Simplified Queries**:
   We create views to abstract the complexity of querying versioned data from multiple tables, making it easier to retrieve both current and historical versions.

# Additional Features

These are some additional features that can add value to the data warehouse and benefit users.

**Backup and Restore Feature**

The *backup process* is managed through the existing versioning system using the `customer_transactions` and `version_metadata` tables. Each dataset version is tracked using `version_id`, `effective_date`, and `end_date`, allowing historical data to be easily referenced.

*Restore Process*: Restoring a version involves:

1. Marking current active records as inactive by updating their `end_date`.
2. Copying data from the target `version_id` and inserting it as a new active version.
3. Recording the action in `version_metadata` for transparency.

**Automated Notifications**

Purpose: Alert users when a new version of the dataset is available, ensuring they are informed promptly for decision-making or analysis.

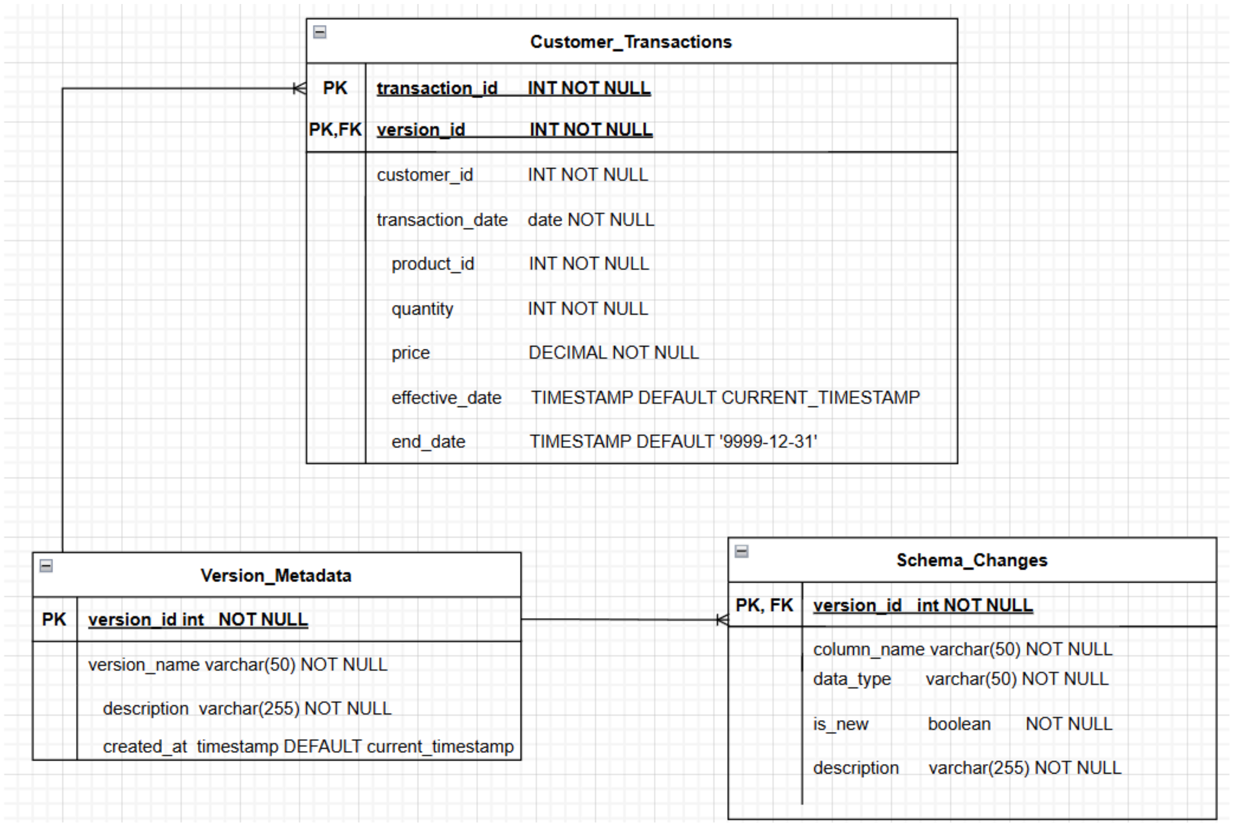
Implementation Approach:

- Integrate a job scheduler (Apache Airflow or a database job scheduler) that runs whenever a new version is added to `version_metadata`.
- Use a messaging service like SMTP email, or Slack API to send notifications.

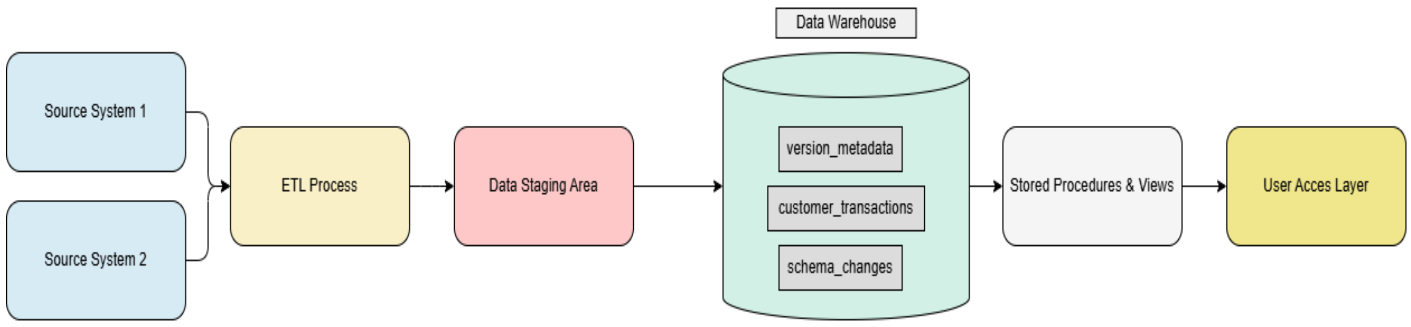
**Performance Dashboard**

Create a dashboard that shows the history of data versioning, such as how many changes were made over time, which tables are most frequently updated, and the size growth of each version.

## ERD (Entity Relationship Diagram)


Customer_Transactions
PK transaction_id INT NOT NULL
PK,FK version_id INT NOT NULL
customer_id INT NOT NULL
transaction_date date NOT NULL
product_id INT NOT NULL
quantity INT NOT NULL
price DECIMAL NOT NULL
effective_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
end_date TIMESTAMP DEFAULT '9999-12-31'
Version_Metadata
PK version_id int NOT NULL
version_name varchar(50) NOT NULL
description varchar(255) NOT NULL
created_at timestamp DEFAULT current_timestamp
Schema_Changes
PK, FK version_id int NOT NULL
column_name varchar(50) NOT NULL
data_type varchar(50) NOT NULL
is_new boolean NOT NULL
description varchar(255) NOT NULL

## Architecture



## Samples Code Snippets

1. Creating version_metadata table
2. Creating new customer_transactions table that includes extra field to track row-level versions
3. Creating schema_changes table to track any changes in the schema (column-level)
4. Insert new version
5. Load Data with Version (Tracking Row-Level)
6. Track schema changes
7. Some example views and stored procedures

```sql
CREATE TABLE version_metadata (
version_id INT PRIMARY KEY,
version_name VARCHAR(50),
created_at   TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
description VARCHAR(255)
);

CREATE TABLE customer_transactions (
    customer_id INT,
    transaction_id INT,
    transaction_date DATE,
    product_id INT,
    quantity INT,
    price DECIMAL(10, 2),
    version_id INT,
    effective_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    end_date TIMESTAMP DEFAULT '9999-12-31',
    PRIMARY KEY (transaction_id, version_id),
    FOREIGN KEY (version_id) REFERENCES version_metadata(version_id)
);

CREATE TABLE schema_changes (
    version_id INT,
    column_name VARCHAR(50),
    data_type VARCHAR(50),
    is_new BOOLEAN,
    change_description VARCHAR(255),
    PRIMARY KEY (version_id, column_name),
    FOREIGN KEY (version_id) REFERENCES version_metadata(version_id)
);

-- Insert New Version
```

```sql
INSERT INTO version_metadata (version_id, version_name, description)
VALUES (2, '2024-11 Monthly Update', 'Added region column and updated
records');

-- Load Data with Version (Tracking Row-Level)
INSERT INTO customer_transactions (customer_id, transaction_id,
transaction_date, product_id, quantity, price, version_id)
VALUES (101, 1001, '2024-11-01', 2001, 3, 150.00, 2);

-- Track Schema Changes
INSERT INTO schema_changes (version_id, column_name, data_type, is_new,
change_description)
VALUES (2, 'region', 'VARCHAR(50)', TRUE, 'Added region column for
geographic tracking');


-- STORED PROCEDURES

-- Retrieve Latest Version

CREATE VIEW current_customer_transactions AS
SELECT *
FROM customer_transactions
WHERE version_id = (SELECT MAX(version_id) FROM version_metadata)
AND end_date = '9999-12-31';

-- Retrieve Specific Historical Version

CREATE PROCEDURE GetCustomerTransactionsByVersion(IN version_input INT)
BEGIN
SELECT *
FROM customer_transactions
WHERE version_id = version_input
AND end_date = '9999-12-31';
 END;

-- List All Available Versions

CREATE PROCEDURE ListAllVersions() BEGIN SELECT version_id, version_name,
created_at, description FROM version_metadata; END;
```

## Links & Resources

- Github Repo
- Data Warehouse Layers
- Customer Transactions sample data (Version 1 & 2)
- The Data Warehouse Toolkit (Book)