

Dimensional Data Modeling with Databricks

Can we use **Databricks** for **data warehousing** and **dimensional modeling**? Absolutely — and I'd argue we *should*! Why? Because Databricks is more than just a platform for big data processing and file-based storage. With **Unity Catalog** and **Delta Lake**, you can implement a **modern Lakehouse architecture** that combines the benefits of a data lake and a data warehouse in a single platform.



Why Does This Matter?

A modern data platform must support:

- **Ad-hoc analysis** and fast access to data through curated **silver layers**

- **Business intelligence (BI)** and reporting needs through **gold layers**

Furthermore, with the rise of **generative AI** tools like **Databricks AI/BI Genie** for **conversational analytics**, it's clear:

There's still a strong need for **structured data models**, and Databricks gives you the tools to build them effectively.

What is Dimensional Modeling?

Dimensional modeling organizes data into **facts** and **dimensions**, optimized for analytics and reporting.

Each **dimension table** has a **primary key** that relates to a **foreign key** in the **fact table**. This structure is often referred to as a **star schema**. In more complex cases, a **snowflake schema** is used, where dimension tables are normalized into related sub-dimensions.

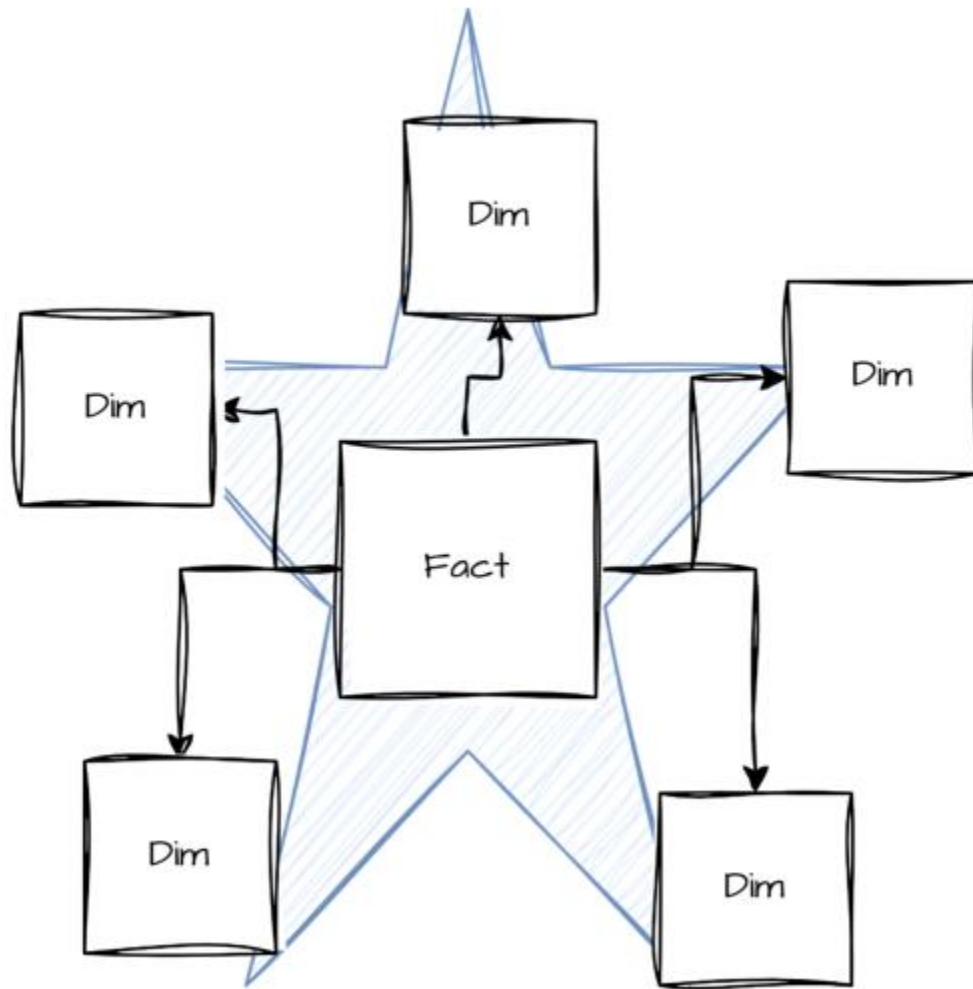
Star Schema

A **star schema** is a classic, denormalized model where a central fact table is connected to multiple dimension tables.

Use cases: Data warehouses, data marts, OLAP systems, reporting tools

Benefits of Star Schema

- Easy for business users to understand
- Excellent performance for **simple, read-heavy queries**
- Fewer joins improve **query speed**
- Ideal for **OLAP models**
- Easy to implement and scale



Star Schema

Snowflake Schema

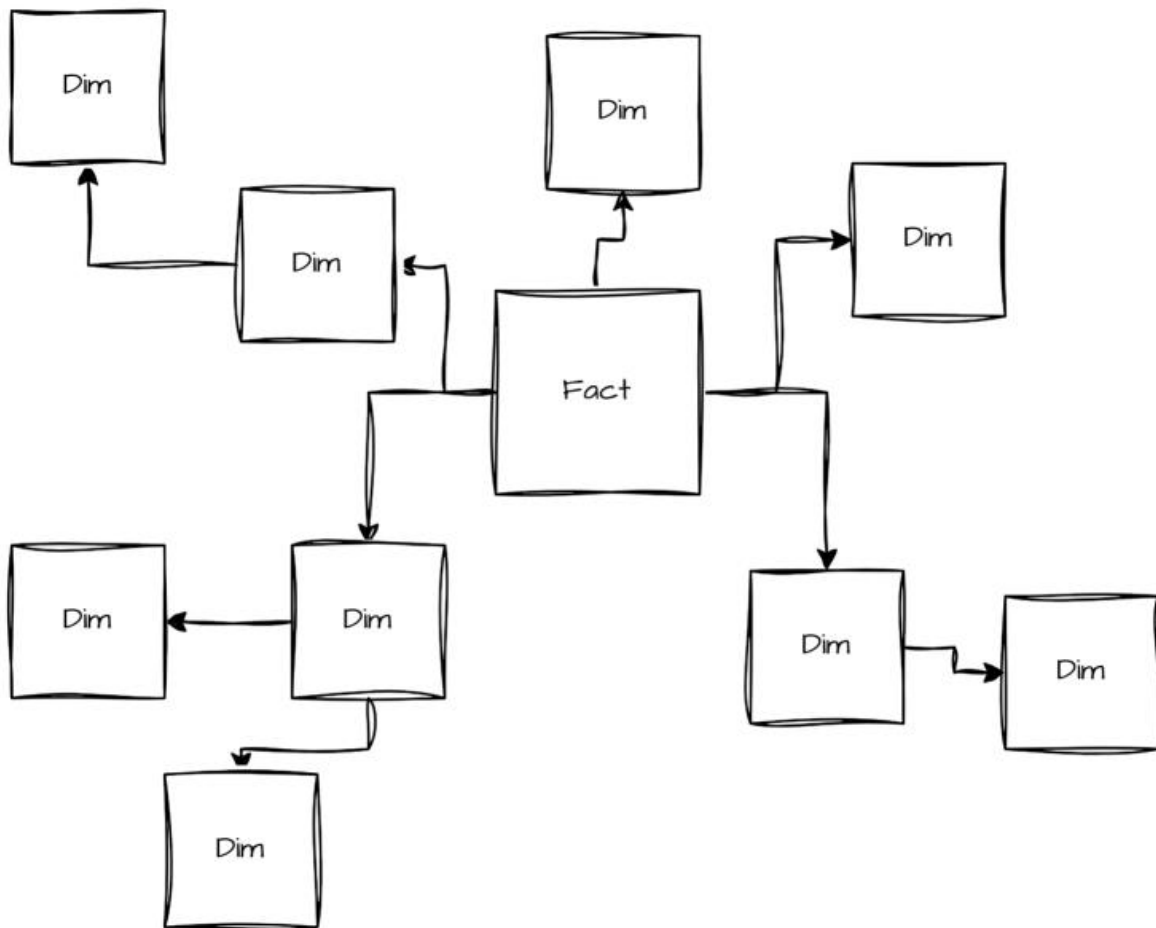
A **snowflake schema** is a normalized variant of the star schema. Dimension tables are broken into related sub-dimensions to reduce redundancy.

Use cases: Complex BI systems, enterprise data warehouses with many hierarchies

Benefits of Snowflake Schema

- Reduces data redundancy
- Improves **data quality and consistency**
- Better suited for **frequent dimension updates**

- Maintains a **normalized, scalable structure**



Snowflake schema

Benefits of Dimensional Modeling

- **Simplifies data consumption** for analysts and business users
- **Boosts performance** with faster, streamlined queries
- Offers **flexibility and scalability** to adapt to evolving requirements
- Enables **data integration** across multiple systems
- Supports **self-service BI** and greater **user adoption**
- Enhances **data governance and security**
- Helps **decouple from volatile source systems**

- Combines multiple source systems under a single, unified model

Dimension Table

A **dimension table** is a core component of **dimensional modeling** that stores descriptive attributes—such as product names, categories, customer details, and geographic locations—which provide context to the numerical data in **fact tables**. These attributes enrich the data and enable more meaningful and actionable analysis.

Each record in a dimension table is uniquely identified by a **surrogate key**, which is a system-generated identifier used internally within the data platform. Although the table may also include a **business key** (e.g., product ID from the source system), it's generally best practice to use a surrogate key for internal references.

This approach is especially useful when integrating data from multiple systems—such as various CRM or ERP instances—where business keys might overlap or conflict. Using a surrogate key ensures **uniqueness and referential integrity** across the model.

Implementing Surrogate Keys in Databricks

In Databricks, surrogate keys are commonly implemented using an **identity column**, which automatically generates a sequential `BIGINT` value for each new row:

```
CREATE TABLE IF NOT EXISTS DimCustomer(  
  
  CustomerSK BIGINT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,  
  
  Name STRING,  
  
  LastName STRING,  
  
  Address STRING,  
  
  InsertDate TIMESTAMP  
  
)
```

Alternatively, you can generate your own surrogate key using a **hashing function** on a business key or a combination of multiple attributes:

```
SELECT

    sha2(concat_ws('|', customer_id, first_name, last_name), 256) AS
customer_sk,

    *

FROM staging_customer;
```

What Does a Dimension Table Contain?

A dimension table typically includes:

- **Descriptive attributes:** e.g., name, type, category
- **Metadata:** e.g., insert date, last update timestamp
- **Surrogate key:** platform-generated unique identifier
- **Optional business key:** source system identifier

For example, a `dim_customer` table may include:

- `customer_sk` (surrogate key)
- `customer_id` (business key)
- `first_name`, `last_name`, `date_of_birth`
- `insert_date`, `update_date`

Tracking metadata like insert or update timestamps is useful for **data quality monitoring** and identifying issues in ETL pipelines.

Dimension Types

Degenerate Dimension:

A degenerate dimension happens when an attribute is stored directly within the fact table rather than in a separate dimension table. For example, a transaction number is often kept in the fact table itself.

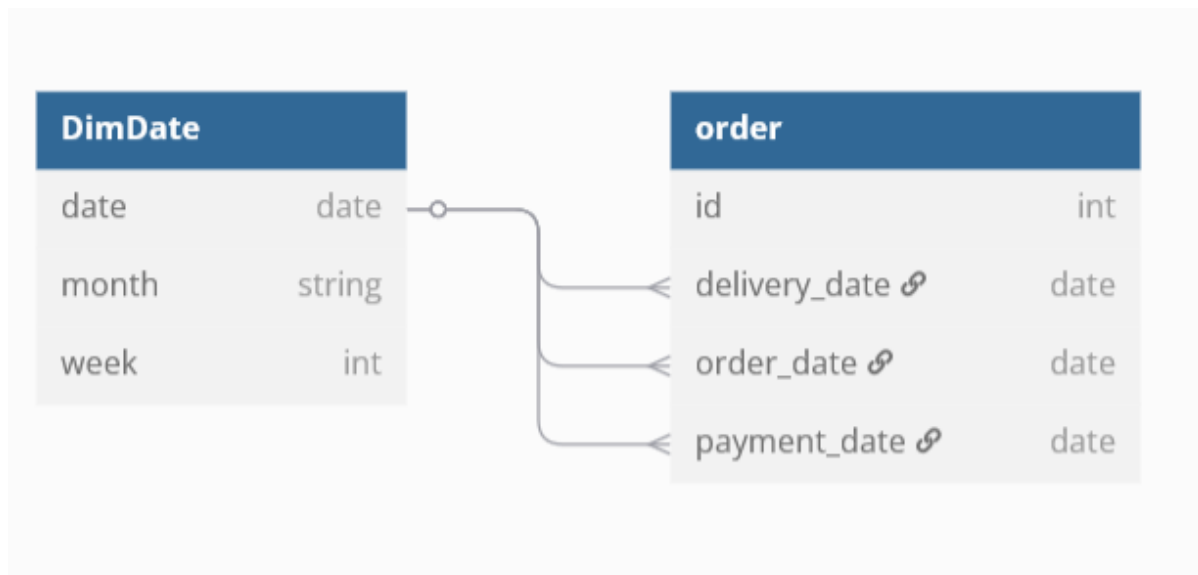
Junk Dimension:

This dimension consists of miscellaneous, low-importance attributes that don't fit neatly into other dimension tables. It usually includes combinations of flags or binary indicators representing various state conditions.

ed	Packed	Shipped	Delivered	Received	Returned
	Y	Y	Y	Y	Y
	Y	Y	Y	Y	Y
	Y	Y	Y	Y	Y
	Y	Y	Y	Y	N
	Y	Y	Y	N	N
	Y	Y	N	N	N
	Y	N	N	N	N
	N	N	N	N	N
	N	N	N	N	N

Role-Playing Dimension:

A role-playing dimension is when a single dimension key appears multiple times in the fact table as different foreign keys. For instance, a date dimension might be used to represent several dates like creation date, order date, and delivery date within the same fact table.



Role-Playing Dimension

Static Dimension:

A static dimension refers to a dimension that rarely changes over time. It is generally loaded once from reference data and doesn't need frequent updates. An example is a list of company branches.

Dimensional Tables Often Contain Hierarchies

In dimensional modeling, **hierarchies** allow users to analyze data at multiple levels of granularity. For example, a reporting tool might display **annual sales**, with the ability to **drill down** into **quarterly**, **monthly**, or **daily** figures for deeper insight.

Common Approaches to Modeling Hierarchies

There are three primary techniques for implementing hierarchies within dimension tables:

1. **Flat (denormalized) hierarchy**—all levels are stored in a single dimension table as separate columns
2. **Snowflake schema**—related dimension tables are normalized into separate entities
3. **Parent-child (self-referencing) hierarchy**—one table references itself to form a recursive relationship

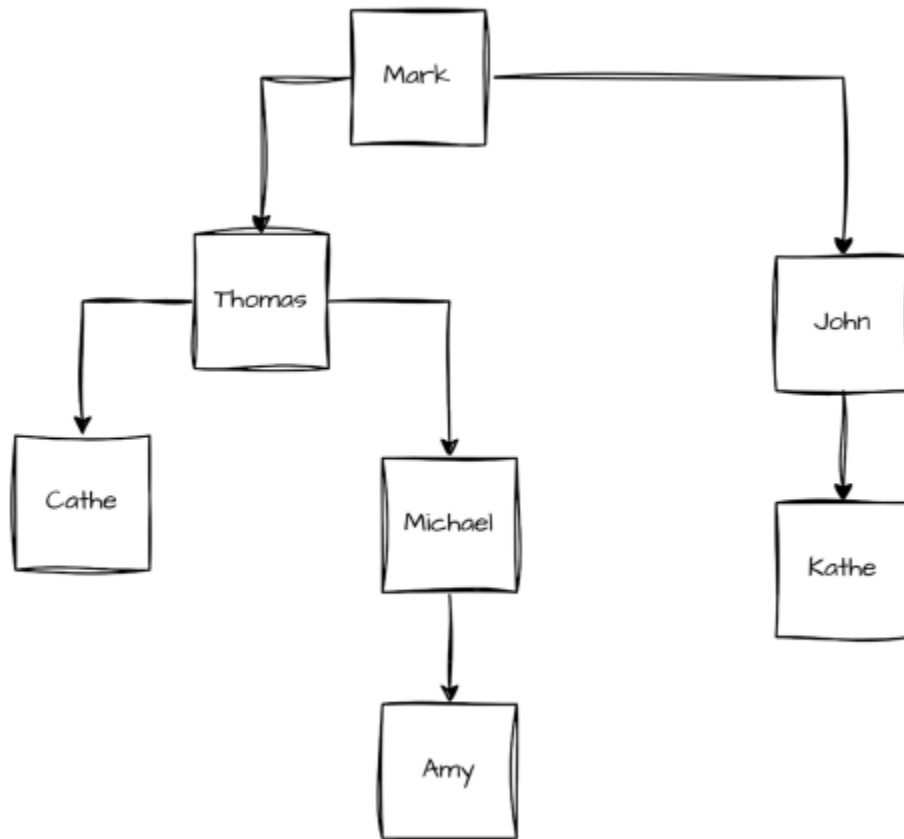
a) Parent-Child Hierarchies (Unbalanced Hierarchies)

A **parent-child hierarchy** is a recursive structure where each row can relate to another row in the same table. These are often **unbalanced**, meaning different branches of the hierarchy may have different depths.

Example: An **employee dimension** where each employee has a `manager_id` pointing to another `employee_id` in the same table.

This type of hierarchy is useful for:

- Organizational structures
- Recursive categories
- Nested reporting groups



```
CREATE TABLE IF NOT EXISTS DimEmployee(  
    emp_id BIGINT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,  
    manager_id BIGINT,  
    name string,  
    surname string  
)
```

b) Balanced Hierarchies (Attribute-Based Hierarchies)

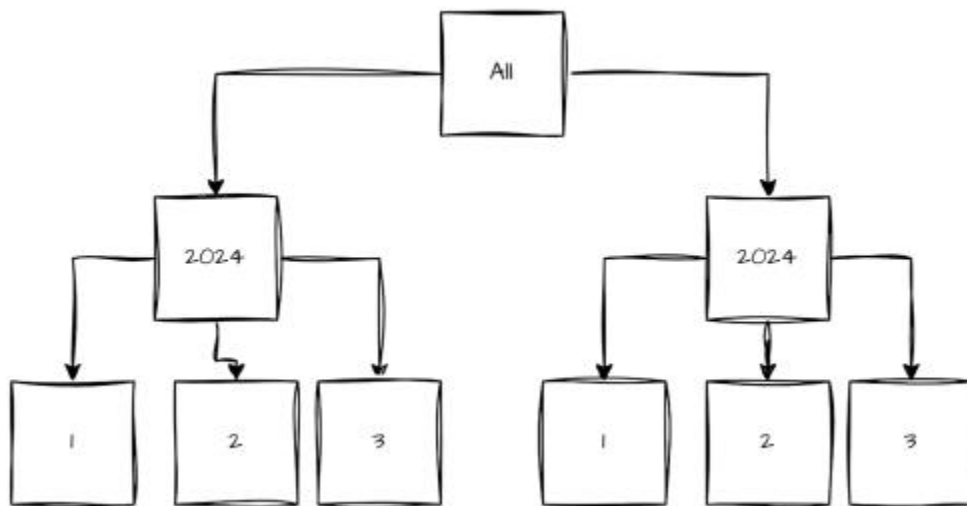
A **balanced hierarchy** has a fixed number of levels, and each level is stored in its own column. This structure is consistent and predictable.

Examples:

- **Geographic hierarchy:** City → Country → Sales Region
- **Calendar hierarchy:** Date → Month → Quarter → Year

These hierarchies are:

- Easy to implement in **denormalized** dimension tables
- Highly optimized for BI tools that support drill-down paths



```
CREATE TABLE IF NOT EXISTS DimRegion(  
    Region_id Bigint,  
    City string,  
    Country string,  
    SalesRegion string  
  
)
```

c) Ragged Hierarchies

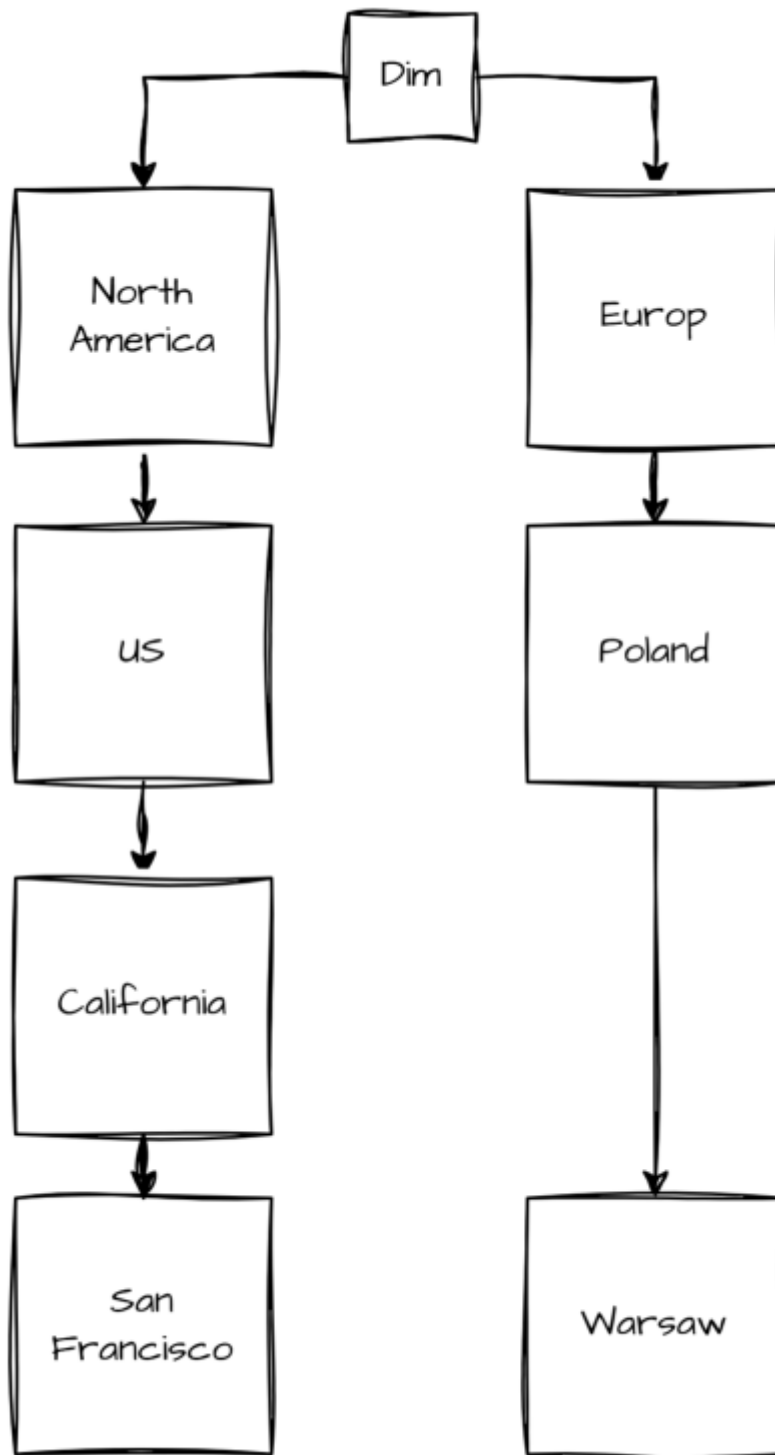
In a **ragged hierarchy**, some lower-level members may **skip levels** in the hierarchy. Like unbalanced hierarchies, branches can have different depths, but the key difference is that **level names remain consistent**, even when levels are skipped.

Example: A **geographic hierarchy** where:

- Some entities go City → State → Country
- Others go directly from City → Country (without a state level)

This pattern is common when:

- Organizational structures differ by region
- Administrative units are inconsistent across geographies



Slowly Changing Dimensions (SCD)

A **Slowly Changing Dimension (SCD)** is a technique in **dimensional modeling** used to manage and track changes in dimension attribute values over time.

SCDs ensure that both **current** and **historical** data are retained, providing accurate context for time-based reporting and analytics.

As business entities (like customers, employees, or products) evolve, their attributes may change—for example, a customer’s address, a product’s category, or an employee’s department. SCD techniques define how these changes are handled in a **dimension table**.

Common SCD Types

There are six types of SCDs, but the three most commonly implemented are:

SCD Type 0

- Changes are **not tracked**
- Records remain unchanged after initial insertion
- Useful for **static** attributes (e.g., Date of Birth)

SCD Type 1—Overwrite

- Updates overwrite existing values
- **No history** is maintained
- Simple and storage-efficient
- Ideal when only the **latest value** is needed

SCD Type 2—Historical Tracking

- **New records** are inserted for each change in attribute value
- Preserves full **change history**
- The original record is marked as inactive or assigned an end date
- Commonly used in **auditable systems, time-based analysis, and BI reporting**








For instance, if John Smith moves to a different city, SCD Type 2 can be used to retain the history of transactions made while he lived in London. This is achieved by inserting

a new record with the updated city and marking the old record, thereby preserving the historical context for reporting.

Implementing SCD Type 2 in Databricks

Databricks provides multiple ways to implement SCD Type 2:

- **Delta Live Tables (DLT)**
- **PySpark with Delta Lake**
- **SQL MERGE statements**

	 customer_sk	 customer_id	 name	 city	 effective_date	 expiry_date	 is_current
1	1	1001	Alice Smith	New York	2022-01-01	2023-05-15	false
2	2	1001	Alice Smith	Boston	2023-05-15	2024-03-10	false
3	3	1001	Alice Johnson	Boston	2024-03-10	9999-12-31	true

SCD2 Example

SCD Type 2 with PySpark and Delta Lake:

```
target_df = DeltaTable.forName(spark, "demo.gold.dim_customer")

(
    target_df
    .alias("dest")
    .merge(
        sourcedf.alias("source"),
        "source.id = dest.id"
    )
    .whenMatchedUpdate(
        set={ "dest.end_date" : current_date() }
    )
    .whenNotMatchedInsertAll()
)
```

```
.execute()  
  
)
```

Fact Table

A **fact table** is the central component of a dimensional model that represents **business events** or **transactions**. It stores **measurable metrics**, such as sales revenue, quantity sold, or website clicks, that provide insights into business performance.

Each record in a fact table corresponds to a **specific event at a defined level of granularity** (e.g., one row per order, per product, per day). The fact table also includes **foreign keys** that link it to related **dimension tables**, enabling rich, multi-dimensional analysis.

Key Components of a Fact Table

A well-designed fact table typically includes the following:

Foreign Keys

- References to related dimension tables (e.g., `customer_id`, `product_id`, `date_id`)
- Enable joins for contextual enrichment

Degenerate Dimensions

- Attributes that don't have their own dimension table but are still useful for filtering or grouping
- Examples: `invoice_number`, `order_id`, `transaction_code`

Measures

- Quantitative data used for analysis
- Examples: `sales_amount`, `quantity`, `discount`, `click_count`

Granularity of a Fact Table

Granularity refers to the level of detail at which data is stored in the fact table. It is one of the most critical design decisions and defines **what a single row represents**—for example:

- One row per order
- One row per product per day

The granularity must be clearly defined and **consistent across all fact rows**, and each joining dimension must align with the fact's level of detail.

Best Practices for Fact Table Design

- **Organize foreign keys at the top** of the schema definition
- **Place degenerate dimensions**
- **List measures at the end** for clarity and consistency
- Avoid mixed-granularity facts in the same table—instead, create separate fact tables for different grains

```
CREATE TABLE IF NOT EXISTS FactTransaction (  
  
    Product_id BIGINT NOT NULL, -- foreign keys to dimension tables  
  
    Sales_id BIGINT NOT NULL,  
  
    Transaction_number BIGINT, -- degenerate dimension keys  
  
    volume BIGINT, -- measures  
  
    UnitPrice DECIMAL(12,4),  
  
    SalesAmount DECIMAL(12,4),  
  
    LastModifiedDateTime TIMESTAMP NOT NULL  
  
    DEFAULT CURRENT_TIMESTAMP() -- delta extract timestamp  
  
);
```

Foreign Key:


```
ALTER TABLE FactTransaction

ADD CONSTRAINT FK_Product_id

FOREIGN KEY (Product_id) REFERENCES DimProduct (Product_id);
```

Fact table types

There are three main types of fact tables, each designed to support different analytical needs:

- **Transaction Fact Tables**—Capture individual events or transactions, such as sales or purchases, with a high level of detail.
- **Periodic Snapshot Fact Tables**—Record data at regular intervals (e.g., daily, monthly), providing a snapshot of the state of business processes over time.
- **Accumulating Snapshot Fact Tables**—Track the lifecycle of a process by capturing key milestone dates and metrics that are updated as the process progresses (e.g., order fulfillment or loan processing).

Factless Fact Table

A factless fact table is a type of fact table that contains no measurable facts or numeric metrics. Instead, it captures events or occurrences, such as student attendance in a class. Although it lacks traditional measures, meaningful analysis can still be performed by counting rows in the table to quantify activity or participation.

Data Modeling in Databricks

Before designing your data model in Databricks, it's important to understand the two primary types of data models used in modern data architecture:

- **Logical Data Model:** Focuses on representing the business entities, relationships, and rules, abstracted from the technical implementation.
- **Physical Data Model:** Translates the logical model into a concrete implementation on a database platform (e.g., Databricks SQL, Delta Lake).

Logical Data Model

A **logical data model** serves as a blueprint that describes *what* data is needed and *how* it relates to the business, without considering physical constraints or platform-specific details.

Steps to Build a Logical Data Model

Identify Entities and Attributes

Define key business entities (e.g., Customer, Product, Transaction) and their relevant attributes (e.g., Name, Price, Date).

Choose Primary Keys (PKs)

Assign unique identifiers to each entity to ensure record uniqueness.

Normalize (or Denormalize) the Model

- Use normalization to reduce redundancy in OLTP systems.
- Consider denormalization in OLAP or analytical systems to simplify joins and improve performance.

Define Relationships

Establish how entities relate to one another (e.g., one-to-many, many-to-many).

Validate Against Business Logic

Review the structure with stakeholders to ensure it reflects actual business rules and workflows.

Physical Data Model

The **physical data model** is a platform-specific implementation of your logical design. It defines *how* the data is stored, structured, and accessed in Databricks or any other data platform.

Steps to Build a Physical Data Model in Databricks

Adapt the Logical Model to the Databricks Environment

Take into account the lakehouse architecture, Unity Catalog, and performance optimization features.

Map Entities to Tables

Convert each logical entity into a physical table in Databricks (Delta format recommended).

Implement Keys and Constraints

Define **primary keys** and **foreign key relationships**, even if they are not enforced, for documentation and BI integration purposes.

Validate Alignment with the Logical Model

Ensure the physical implementation accurately supports the original business requirements.

Define Data Types

Assign appropriate data types to each column (`STRING`, `INT`, `DECIMAL`, `TIMESTAMP`, etc.) to support data integrity and query performance.

Logical vs. Physical

Aspect	Logical Model	Physical Model (Databricks)
Entity	Customer	dim_customer (Delta Table)
Attribute	First Name, Last Name, Email	first_name STRING, email STRING
Primary Key	Customer ID	customer_id BIGINT
Relationship	Customer ↔ Order (1:N)	Foreign Key: order.customer_id

Data Modeling Tools

Choosing the right tool is a critical first step when starting your data modeling journey. A good data modeling tool helps you **visualize your model**, **manage versions as code**, and even **generate DDL scripts** for seamless implementation.

GUI-Based Tools

- **DBviewer**
A user-friendly SQL client with a graphical interface that allows quick visualization and editing of database schemas.
- **Erwin Data Modeler**
A powerful, enterprise-grade GUI tool for designing, visualizing, and generating complex data models.
Databricks integrates well with Erwin, enabling smooth modeling and deployment on the Lakehouse platform.
[Learn more here](#)

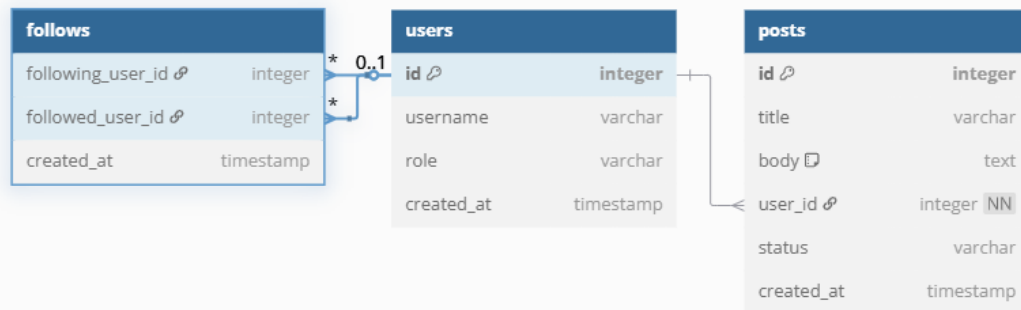
Web-Based Tools

- **Diagrams.net**

A free, browser-based diagramming tool perfect for creating quick and customizable data models and flowcharts.

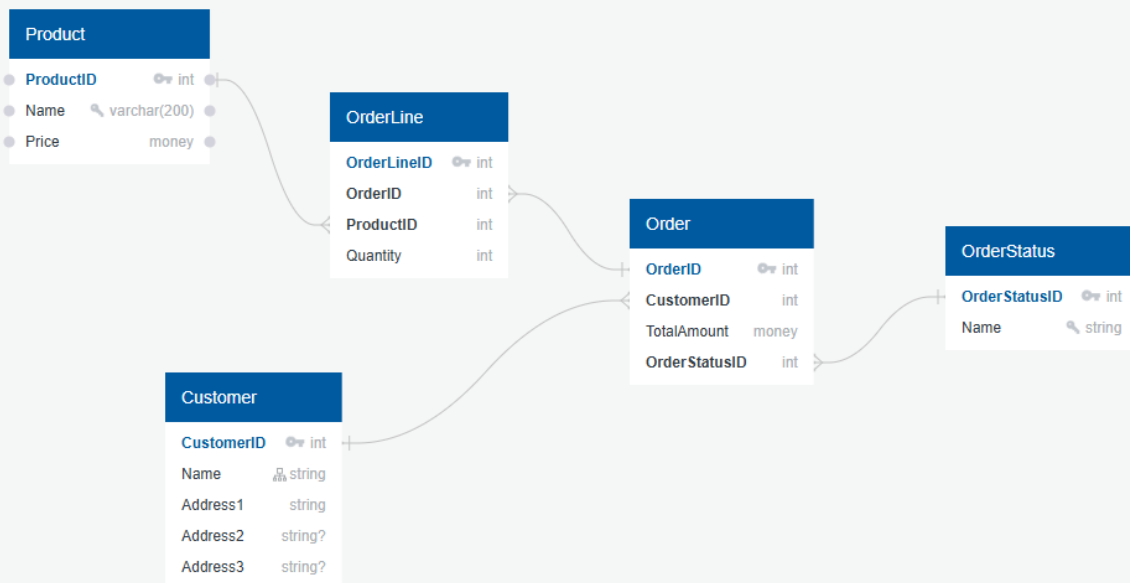
- dbdiagram.io

A web-based diagramming platform that uses **DBML (Database Markup Language)** to define and visualize your data models programmatically. Visit dbdiagram.io



- **QuickDBD (Quick Database Diagrams)**

An online tool that lets you **write your data model using simple text expressions**, which it instantly converts into visual diagrams. This accelerates collaboration and documentation.



Why Use Data Modeling Visualization?

Facilitates **collaboration** between data engineers, analysts, and business users by providing a clear, visual representation of data structures. Enhances **documentation quality** and keeps it aligned with evolving data models. Supports **dependency analysis**, helping teams understand relationships and impacts of changes. Simplifies the challenge of designing models that accurately meet complex business requirements.

How to Use Data from the Data Model in Databricks

Once your data model is ready, the next step is to **consume and analyze the data** efficiently. Databricks offers several options to query and work with data models using familiar SQL interfaces and integrations with BI tools.

Querying Data Using SQL

- **SQL Editor and Serverless SQL Warehouses**
Databricks provides a powerful SQL Editor with Serverless SQL Warehouses (also called Serverless Clusters) that let you run ad-hoc queries directly against your data model. This is ideal for quick exploration and analysis without managing infrastructure.
- **Notebooks and All-Purpose Clusters**
If you prefer working within Databricks notebooks, you can execute SQL commands using the `%sql` magic command. This option is especially useful if you want to combine SQL with other languages like Python or Scala in the same notebook.
- **Familiarity with SQL Users**
If your background is with platforms like SQL Server, Oracle, BigQuery, or Snowflake, querying in Databricks via SQL will feel familiar, allowing you to leverage your existing skills without needing to learn PySpark immediately.



The screenshot shows a Databricks SQL interface. At the top, there is a blue button labeled 'Run (1000)' with a dropdown arrow. To its right, the text 'dbw_demo_dev_001.' is followed by a 'default' dropdown menu. Further right, it says 'New SQL editor: OFF' with a dropdown arrow. Below this, a SQL query is entered in a text area:

```
1 select *
2 from
3 demo.data.customer_raw
```

External SQL Clients

You can also query Databricks data models using external SQL clients such as **DbViewer** or any JDBC/ODBC-compatible tool. This flexibility lets you use your preferred query tools and integrate Databricks into existing workflows.

Integration with Business Intelligence Tools

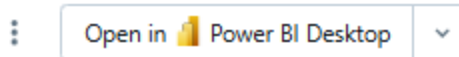
Databricks integrates seamlessly with popular BI platforms like **Power BI**, enabling data analysts and business users to visualize and interact with data models effortlessly.

Publishing a Databricks Dataset to Power BI:

1. Navigate to the desired **schema** or **table** in the Unity Catalog within Databricks.
2. Click “**Use with BI tools**” or “**Open in a dashboard.**”
3. Select “**Publish to Power BI workspace.**”
4. Log in to your Power BI account, then choose your workspace and dataset options.
5. Click “**Publish to Power BI.**”

You can connect via two modes:

- **Import Mode:** Data is imported and stored in Power BI.
- **Direct Query:** Queries are sent live to Databricks using Serverless SQL Warehouses, ensuring up-to-date data without duplication.



Summary

A well-designed data model does more than just answer business questions—it provides an intuitive structure that streamlines report building across tools like Power BI, Tableau, Qlik, and Databricks Dashboards. Furthermore, with the rise of generative AI, new possibilities emerge to **automate report creation** and enable natural language interaction with data, making it easier than ever to uncover valuable insights quickly and efficiently.