What is Data Modelling?

Data modelling is the process of defining and structuring data to be stored, managed, and used efficiently within a database or data system. It involves creating **conceptual, logical, and physical models** that represent data relationships, rules, and constraints.

Types of Data Models

1. **Conceptual Data Model:** High-level view that defines business entities and relationships. Used for business stakeholders.
2. **Logical Data Model:** More detailed, specifying attributes, keys, and relationships without focusing on physical storage.
3. **Physical Data Model:** Defines how data is stored in a database, including tables, columns, indexes, partitions, and data types.

Key Elements of Data Modelling

- **Entities** (e.g., Customer, Order)
- **Attributes** (e.g., Name, Order Date)
- **Relationships** (e.g., One-to-Many, Many-to-Many)
- **Primary & Foreign Keys** (for unique identification and referential integrity)
- **Normalization & Denormalization** (to optimize storage and performance)

Importance of Data Modelling

- Ensures **data consistency, accuracy, and integrity**
- Improves **database performance and scalability**
- Helps in **better decision-making and reporting**
- Facilitates **data governance and compliance**

Different Types of Data Modelling Techniques

Data Modelling techniques help design and structure data for efficient storage, retrieval, and analysis. The three primary types of data Modelling are **Conceptual, Logical, and Physical** models, but different techniques exist based on use cases.

---

1. Hierarchical Data Model

- Organizes data in a **tree-like structure** with parent-child relationships.
- Each parent node can have multiple child nodes, but each child has only one parent.
- Used in **IBM's Information Management System (IMS)** and old **mainframe databases**.

Example:

Company

├── Department A

│ ├── Employee 1

│ ├── Employee 2

├── Department B

├── Employee 3

Use Case: Legacy systems, XML databases.

---

2. Network Data Model

- Similar to the hierarchical model but allows **many-to-many relationships**.
- Uses **graph-based structures** with records connected through links.
- More flexible than the hierarchical model.

Example:

Student → (Enrolled In) → Course

Course → (Has) → Instructor

Instructor → (Teaches) → Student

Use Case: Telecommunications, Banking, Early DBMS like IDMS.

---

3. Relational Data Model (RDM)

- Stores data in **tables (relations)** with rows (records) and columns (attributes).
- Uses **Primary Keys (PK)** and **Foreign Keys (FK)** to establish relationships.
- Based on **Structured Query Language (SQL)**.

Example:

Customers (Customer_ID, Name, Email)

Orders (Order_ID, Customer_ID, Amount)

Use Case: Traditional databases (MySQL, PostgreSQL, Oracle, SQL Server).

---

4. Entity-Relationship Model (ER Model)

- Visual representation of entities, attributes, and relationships using **ER diagrams**.
- Helps in conceptual database design before actual implementation.

- Defines **one-to-one (1:1), one-to-many (1:M), and many-to-many (M:M) relationships**.

Example:

Customer (Customer_ID) ----> (Places) ----> Order (Order_ID)

Use Case: Initial database design, Data Warehousing.

---

5. Dimensional Data Model (Used in Data Warehouses)

- Optimized for fast querying and reporting.
- Uses **Fact Tables** (numerical values) and **Dimension Tables** (descriptive attributes).
- Includes **Star Schema** (denormalized) and **Snowflake Schema** (normalized).

Example:

Sales_Fact (Date_ID, Product_ID, Store_ID, Sales_Amount)

Dimension Tables: Date_Dim, Product_Dim, Store_Dim

Use Case: Data Warehouses (AWS Redshift, Snowflake, BigQuery).

---

6. Object-Oriented Data Model

- Stores data as **objects** (like in programming languages).
- Objects contain attributes and methods (functions).
- Used in **Object-Oriented Databases (OODB)**.

Example:

```
class Customer:
  def __init__(self, name, age):
    self.name = name
    self.age = age
```

Use Case: CAD Systems, Multimedia Databases.

---

7. Document-Oriented Model (NoSQL)

- Data is stored as **JSON or BSON documents** instead of tables.
- Schema-less and flexible structure.
- Used in **MongoDB, CouchDB**.

Example (MongoDB JSON Document):

```
{
 "customer_id": 101,
 "name": "John Doe",
 "orders": [
  { "order_id": 1, "amount": 500 },
  { "order_id": 2, "amount": 300 }
 ]
}
```

Use Case: Big Data, Web Applications, NoSQL Databases.

---

8. Graph Data Model

- Stores data in **nodes and edges** (relationships).
- Ideal for **complex relationships** like social networks and fraud detection.
- Used in **Neo4j, Amazon Neptune, ArangoDB**.

Example (Social Network Graph):

(User1) ----[Follows]----> (User2)

(User2) ----[Follows]----> (User3)

Use Case: Social Networks, Recommendation Engines, Fraud Detection.

Comparison of Data Modelling Techniques

Technique	Best For	Example Databases
Hierarchical	Legacy Systems, XML Databases	IBM IMS, Windows Registry
Network	Complex Relationships, Telecom	IDMS, CA-Datacom
Relational (RDM)	Traditional Applications	MySQL, PostgreSQL, Oracle
ER Model	Conceptual Database Design	Any Relational DB
Dimensional	Data Warehousing, BI	AWS Redshift, Snowflake
Object-Oriented	CAD, Multimedia Databases	db4o, ObjectDB
Document-Based	NoSQL, JSON Data Storage	MongoDB, CouchDB
Graph-Based	Social Networks, AI	Neo4j, Amazon Neptune

What is Dimensional Modelling?

Dimensional modelling is a **data design technique** used in **data warehouses and business intelligence (BI) systems** to optimize data retrieval for analytical and reporting purposes. It structures data into **fact and dimension tables** to support fast query performance and easy business interpretation.

Why Use Dimensional Modelling?

✅ **Faster Query Performance** (Denormalized structure reduces joins)
✅ **Simplifies Reporting & Analytics** (Intuitive for business users)
✅ **Supports Aggregations & Drill-Downs** (Summarized and detailed analysis)
✅ **Scalability** (Handles large datasets efficiently in Redshift, Snowflake, etc.)

What are Facts in Dimensional Modelling?

In **dimensional modelling**, **facts** are the **measurable, quantitative data** stored in a **fact table**. Facts represent business events or transactions and are used for reporting and analytics.

Key Characteristics of Facts

✅ **Numeric Values** (e.g., Sales Amount, Order Count)
✅ **Foreign Keys** linking to **dimension tables**
✅ **Granularity Level** (defines the detail level, e.g., daily sales vs. monthly sales)

Why Are Facts Important?

✅ Drive **Business Insights & Reporting**
✅ Enable **Aggregations, Drill-Downs, and Trends**
✅ Support **Decision-Making & Forecasting**

What is Additivity in Facts?

Additivity in facts refers to how fact values behave when aggregated across different dimensions. It determines whether a fact can be **summed, averaged, or not aggregated at all** across various business perspectives.

Types of Additivities in Facts

1. **Additive Facts**

- Can be **summed across all dimensions** (time, location, product, etc.).
- **Example:** Sales_Amount, Quantity_Sold
- ✅ Can be aggregated across Date, Store, and Product.

Example Query (AWS Redshift / SQL):

```
SELECT Store_ID, SUM(Sales_Amount) AS Total_Sales
FROM Sales_Fact
GROUP BY Store_ID;
```

---

2. **Semi-Additive Facts**

- Can be **summed across some dimensions** but **not others**.
- **Example:** Account_Balance, Inventory_Level
- ✅ Can be summed across stores.
- ❌ Cannot be summed across time (e.g., summing account balances across multiple days is incorrect).

Example Query (AWS Redshift / SQL):

```
SELECT Store_ID, AVG(Account_Balance) AS Avg_Balance
FROM Bank_Fact
GROUP BY Store_ID;
```

---

3. **Non-Additive Facts**

- **Cannot be aggregated** directly.
- Typically, ratios, percentages, or calculated metrics.
- **Example:** Profit_Margin (%), Discount_Rate, Conversion_Rate.
- ✅ Can be averaged but not summed.

Example Query (AWS Redshift / SQL):

```
SELECT Product_ID, AVG(Profit_Margin) AS Avg_Margin
FROM Sales_Fact
GROUP BY Product_ID;
```

Comparison of Additivity Types

Type	Example Facts	Aggregation Across Time?	Aggregation Across Other Dimensions?
Additive	Sales_Amount, Quantity_Sold	✅ Yes	✅ Yes
Semi-Additive	Account_Balance, Inventory_Level	❌ No	✅ Yes
Non-Additive	Profit_Margin (%), Discount_Rate	❌ No	❌ No (only averages work)

Why is Additivity Important?

✅ Ensures **correct aggregations** in reporting.
✅ Helps in designing **fact tables** for BI tools like **Power BI, AWS Redshift, and Snowflake**.
✅ Improves **query performance and accuracy** in data warehouses.

Handling NULLs in Facts in Dimensional Modelling

In **Fact tables**, NULL values can occur due to **missing, unknown, or inapplicable data**. Proper handling of NULLs is important for **data integrity, query performance, and accurate reporting**.

Possible Reasons for NULLs in Fact Tables

1. **Delayed Transactions**: A sale is recorded, but revenue data is not yet available.
   - Example: Sales_Amount = NULL when payment is pending.
2. **Data Processing Issues**: Data ingestion failure or missing values from the source system.
3. **Factless Fact Tables**: Some tables track events without numeric facts.
   - Example: A Student_Course_Enrollment table may track course signups without grades yet.
4. **Irrelevant Metrics**: Some facts are not applicable in certain contexts.
   - Example: In an e-commerce order table, Discount_Amount might be NULL for full-priced items.

Strategies to Handle NULLs in Fact Tables

✅ **Substituting with Defaults**

- Replace NULLs with **0 for numeric facts** if it makes logical sense.

- **Example:** NULL → 0 for Sales_Amount in a fact table.

SELECT COALESCE(Sales_Amount, 0) AS Sales_Amount FROM Sales_Fact;

✅ **Forward Filling or Backward Filling**

- Fill missing values using previous or next available data.
- Useful for **inventory or stock-level facts**.

✅ **Excluding NULLs from Aggregations**

- Ensure NULLs don't affect SUM, AVG, etc.
- Example: Using SUM(Sales_Amount) in Redshift or Power BI automatically ignores NULLs.

✅ **Using Conditional Aggregation**

SELECT Store_ID, SUM(COALESCE(Sales_Amount, 0)) AS Total_Sales

FROM Sales_Fact

GROUP BY Store_ID;

✅ **Factless Fact Tables (If No Numeric Facts Exist)**

- If a fact table is just tracking events (e.g., student course enrollment), NULLs are normal.

Best Practices for NULL Handling

🔷 **Define Business Rules for NULLs** (Decide when NULLs are expected and how to handle them).
🔷 **Ensure Data Cleaning in ETL Pipelines** (Use AWS Glue, Apache Airflow, etc.).
🔷 **Avoid NULLs in Aggregated Queries** (Use COALESCE() or IFNULL()).
🔷 **Use Default Values if NULLs Are Not Valid** (e.g., 0 for missing sales).

Year-To-Date (YTD) Facts in Dimensional Modelling

Year-To-Date (YTD) Facts represent the **cumulative total of a fact** (e.g., sales, revenue, expenses) from the beginning of the year up to a specific date. YTD calculations help in **trend analysis, performance tracking, and business forecasting**.

YTD Calculation in SQL (AWS Redshift / Snowflake)

Example Fact Table (Sales_Fact)

Date	Store_ID	Product_ID	Sales_Amount
2024-01-01	1	101	500
2024-01-02	1	101	300
2024-02-01	1	102	700
2024-03-01	1	103	600

---

1. YTD Using SUM() OVER() Window Function

```
SELECT
   Date,
   Store_ID,
   SUM(Sales_Amount) OVER (PARTITION BY Store_ID
                ORDER BY Date
                ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS YTD_Sales
FROM Sales_Fact;
```

✅ This computes **cumulative sales** from the start of the year **per store**.

---

2. YTD Using SUM() with DATE_TRUNC()

```
SELECT
   Store_ID,
   DATE_TRUNC('year', Date) AS Year_Start,
   Date,
   SUM(Sales_Amount) OVER (PARTITION BY Store_ID, DATE_TRUNC('year', Date)
                ORDER BY Date) AS YTD_Sales
FROM Sales_Fact;
```

✅ Ensures YTD calculations reset **every new year**.

---

3. YTD Aggregation for Reports

```
SELECT
  Store_ID,
  SUM(Sales_Amount) AS YTD_Sales
FROM Sales_Fact
WHERE Date BETWEEN DATE_TRUNC('year', CURRENT_DATE) AND CURRENT_DATE
GROUP BY Store_ID;
```

✅ Retrieves **YTD Sales for the current year**.

Best Practices for YTD Facts

✓ **Use Window Functions** for efficient SQL-based YTD calculations.
✓ **Partition Data Correctly** to avoid unnecessary recalculations.
✓ **Optimize BI Queries** by pre-aggregating YTD values in ETL pipelines.
✓ **Ensure Performance** by indexing the Date column in Redshift, Snowflake, etc.

Types of Fact Tables in Dimensional Modelling

A **fact table** stores **measurable business data** (e.g., sales, revenue, quantity sold) and foreign keys linking to **dimension tables** (e.g., time, product, customer). Different types of fact tables are used based on business needs.

---

1. Transactional Fact Table

✅ **Captures individual transactions at the most detailed level.**
✅ Each row in the table represents a single event, such as an **order, sale, or payment**.
✅ The data is **insert-only**, meaning that records are not updated after being written.
✅ Typically includes **foreign keys** to dimension tables like **time, product, store, and customer**, along with **measurable facts** like Sales_Amount and Quantity_Sold.

💡 **Example Use Cases:**

- An **e-commerce sales table** tracking each order made on a website.
- A **banking transaction table** storing deposits and withdrawals.

Example: Sales Fact Table

Date	Store_ID	Product_ID	Customer_ID	Sales_Amount	Quantity_Sold
2024-02-01	1	101	501	500	5
2024-02-01	2	102	502	700	7

---

2. Periodic Snapshot Fact Table

✅ **Captures aggregated data at regular time intervals** (daily, weekly, monthly, etc.).
✅ Provides a historical view of performance and helps track **trends** over time.
✅ Instead of tracking individual transactions, it **aggregates** them into periodic summaries.
✅ The data is typically not updated but **new snapshots** are added periodically.

💡 **Example Use Cases:**

- A **daily store sales summary** table in retail.
- A **monthly inventory level report** in supply chain management.

Example: Daily Sales Snapshot

Date	Store_ID	Total_Sales	Total_Orders	Total_Customers
2024-02-01	1	10,000	50	40
2024-02-02	1	12,500	60	50

👉 **Advantage:** Faster querying for reports because data is pre-aggregated.

---

3. Accumulating Snapshot Fact Table

✅ **Tracks the full lifecycle of a business process, with multiple status updates over time.**
✅ Unlike transactional fact tables (insert-only), accumulating snapshots **update records** as processes move through different stages.
✅ The table contains **timestamps for different milestones** (e.g., Order Date, Payment Date, Shipping Date, Delivery Date).
✅ If an order is completed, its corresponding record in the fact table is updated.

💡 **Example Use Cases:**

- **Order fulfilment tracking** (order placed → payment received → shipped → delivered).
- **Loan processing tracking** (loan application submitted → approved → disbursed).

Example: Order Processing Fact Table

Order_ID	Customer_ID	Order_Date	Payment_Date	Shipping_Date	Delivery_Date
1001	501	2024-02-01	2024-02-02	2024-02-03	2024-02-05
1002	502	2024-02-02	NULL	NULL	NULL

👉 **Advantage:** Useful for tracking the **progress of a business workflow** and identifying delays.

---

4. Factless Fact Table

✅ **Contains only foreign keys to dimensions, without any measurable numeric facts.**
✅ Used for tracking **events or conditions** that do not have associated numerical metrics.
✅ Helps in identifying **patterns, participation, and event tracking**.
✅ Often used in many-to-many relationships, such as tracking student-course enrolments.

💡 **Example Use Cases:**

- **Student-course enrolment** (which students are enrolled in which courses).
- **Employee attendance tracking** (which employees attended a specific training session).

Example: Student Course Enrolment

Student_ID	Course_ID	Enrollment_Date
201	101	2024-01-10
202	102	2024-01-12

👉 **Advantage:** Even without numerical values, this type of fact table **helps in analytical reporting** (e.g., how many students enrolled in a course).

Comparison of Fact Table Types

Fact Table Type	Granularity	Data Update Type	Use Case Example
Transactional Fact Table	Lowest (each event)	Insert-only	Online orders, banking transactions
Periodic Snapshot Fact Table	Aggregated (daily, monthly)	Insert-only (new snapshots)	Daily sales, monthly stock levels
Accumulating Snapshot Fact Table	Lifecycle tracking	Updates as milestones occur	Order fulfilment, loan processing
Factless Fact Table	Event tracking	Insert-only	Student-course enrolment, employee attendance

---

Best Practices for Designing Fact Tables

✓ **Choose the right fact table type** based on the business need.
✓ **Use correct granularity** to ensure optimized data storage and querying.
✓ **Partition large fact tables** (e.g., by date) to improve performance in **AWS Redshift, Snowflake, or BigQuery**.
✓ **Index foreign keys** to improve query efficiency.
✓ **Use aggregations in snapshot tables** to optimize report performance.

Steps in Designing Fact Tables

Designing a **fact table** requires careful planning to ensure **efficient storage, fast query performance, and accurate business insights**. Below are the key steps to follow:

---

1. Identify Business Process to Analyse

✅ **Understand the business requirements** and determine what process needs analysis.
✅ Examples:

- "How many sales occurred in each store per day?"
- "What is the order processing time from placement to delivery?"
  ✅ This step helps in determining **what data should be captured** and analysed.

---

2 Define the Grain (Granularity) of the Fact Table

✅ **Decide the level of detail** for each record in the fact table.
✅ Common granularity options:

- **Transaction-level grain** (each row represents an individual sale, order, or event).
- **Daily snapshot grain** (each row represents total sales per store per day).
- **Monthly snapshot grain** (aggregated sales per store per month).
  ✅ Example:
- **Fine-grained:** Each sales transaction per product per store.
- **Coarse-grained:** Total daily sales per store.

💡 **Choosing the right granularity** ensures optimal performance and flexibility in analysis.

---

3. Identify the Measures (Facts) to Capture

✅ **Select the numeric metrics that will be analysed and reported.**
✅ Example facts in a Sales Fact Table:

- Sales_Amount (Revenue from the transaction).
- Quantity_Sold (Number of items sold).
- Discount_Applied (Any discount given on the order).
  ✅ Measures should be **additive, semi-additive, or non-additive**, based on reporting needs.

💡 **Ensure that all required measures are included while avoiding unnecessary metrics.**

---

4. Identify the Dimensions to Link

✅ **Determine the descriptive attributes (dimensions) that will provide context to the facts.**
✅ Examples of **common dimensions**:

- **Date Dimension:** Time-based analysis (e.g., daily, weekly, monthly reports).
- **Product Dimension:** Information about the product (e.g., category, price, brand).
- **Customer Dimension:** Customer details (e.g., region, age, gender).
- **Store Dimension:** Store details (e.g., store location, type, manager).

💡 **Dimension tables help in slicing and dicing the fact data for detailed analysis.**

---

5. Establish Relationships (Foreign Keys) Between Fact and Dimensions

✅ Each fact record should reference **foreign keys** to related dimension tables.
✅ Example Fact Table Schema:

Date_ID	Store_ID	Product_ID	Customer_ID	Sales_Amount	Quantity_Sold
20240201	1	101	501	500	5
20240202	2	102	502	700	7

✅ The **Date_ID**, **Store_ID**, **Product_ID**, and **Customer_ID** are **foreign keys** linking to their respective dimension tables.

💡 **Foreign key relationships enable powerful analytical queries and efficient joins.**

---

6. Handle Fact Table Additivity

✅ Decide how the measures should behave in **summarized reports**:

- **Additive Facts:** Can be summed across all dimensions (e.g., Sales_Amount, Quantity_Sold).
- **Semi-Additive Facts:** Can be summed across some dimensions but not all (e.g., Account Balance can be summed across stores but not across time).
- **Non-Additive Facts:** Cannot be summed at all (e.g., Profit Margin Percentage).

💡 **Understanding additivity helps in designing meaningful aggregations.**

---

7. Handle Missing Data and Null Values

✅ Determine how to handle **missing values or nulls** in the fact table.
✅ Common strategies:

- Replace NULL values with **default values** (e.g., 0 for missing sales data).
- Use **factless fact tables** if no numerical data is available (e.g., tracking student course enrolments).

💡 **Proper handling of missing data ensures accurate reporting.**

---

8. Partitioning and Indexing for Performance

✅ Optimize the fact table for **fast query performance** using:

- **Partitioning** (e.g., partition by Date_ID for faster filtering).
- **Indexing** (e.g., index on Store_ID or Customer_ID for faster joins).
  ✅ Consider **columnar storage formats** (e.g., ORC, Parquet in AWS Glue) for big data performance.

💡 **Efficient storage techniques reduce query response time and improve performance.**

---

9. Implement ETL (Extract, Transform, Load) Process

✅ Design an **ETL pipeline** to:

- Extract data from source systems (e.g., SQL Server, APIs, mainframe).
- Clean and transform the data (e.g., handling missing values, formatting timestamps).
- Load the data into the fact table in a **data warehouse** (AWS Redshift, Snowflake, BigQuery).

💡 **A well-designed ETL pipeline ensures data consistency and accuracy.**

---

10. Test and Validate the Fact Table

✅ Run **sample queries** to verify correctness:

- Ensure **data joins correctly** with dimension tables.
- Validate **aggregations** (e.g., summing Sales_Amount by Store_ID).
  ✅ Check **data integrity**:
- Ensure no orphaned foreign keys (fact table rows should reference valid dimension records).
- Confirm data accuracy against source systems.

💡 **Thorough testing prevents incorrect reporting and analytics.**

Example: Sales Fact Table Schema in SQL

```
CREATE TABLE Sales_Fact (
    Date_ID       INT,
    Store_ID      INT,
    Product_ID    INT,
    Customer_ID   INT,
    Sales_Amount  DECIMAL(10,2),
    Quantity_Sold INT,
    Discount_Applied DECIMAL(5,2),
    FOREIGN KEY (Date_ID) REFERENCES Date_Dim(Date_ID),
    FOREIGN KEY (Store_ID) REFERENCES Store_Dim(Store_ID),
    FOREIGN KEY (Product_ID) REFERENCES Product_Dim(Product_ID),
    FOREIGN KEY (Customer_ID) REFERENCES Customer_Dim(Customer_ID)
) PARTITIONED BY (Date_ID);
```

💡 **This schema efficiently organizes and optimizes the fact table for querying.**

Final Summary: Steps in Designing a Fact Table

Step	Description
1. Identify Business Process	Determine what business event to track.
2. Define the Grain (Granularity)	Decide the level of detail per record.
3. Identify the Measures (Facts)	Choose numeric values to store (e.g., Sales_Amount).
4. Identify the Dimensions	Select attributes for analysis (e.g., Product, Date).
5. Establish Foreign Keys	Link fact table to dimension tables.
6. Handle Additivity	Define if facts are **additive, semi-additive, or non-additive**.
7. Handle Missing Data	Address NULL values properly.
8. Partition and Index Data	Optimize performance for queries.
9. Implement ETL Process	Extract, transform, and load data.
10. Test and validate	Ensure data integrity and correctness.

Surrogate Keys in Data Warehousing & Dimensional Modelling

What is a Surrogate Key?

A **surrogate key** is a system-generated unique identifier (typically an integer) used as the **primary key** in a table, especially in **dimension tables** in a data warehouse. Unlike **natural keys**, which come from the source system (e.g., Customer ID, Product Code), surrogate keys are independent of business data and have no real-world meaning.

Why Use Surrogate Keys?

✅ **Uniqueness & Integrity:**

- Ensures each record has a unique, immutable identifier.
- Helps prevent duplicates or inconsistencies caused by changes in business data.

✅ **Handles Slowly Changing Dimensions (SCDs):**

- When business data changes (e.g., a customer moves to a new city), a new surrogate key allows tracking history while keeping the original data.

✅ **Prevents Dependence on Source Systems:**

- Source keys (e.g., Customer ID) can be **reused, change format, or be null**—causing issues in a data warehouse.
- Surrogate keys ensure consistency even when source systems change.

✅ **Optimized Joins & Performance:**

- Integer-based surrogate keys are **faster to join** than long or complex natural keys (e.g., alphanumeric customer codes).
- Reduces query time in large fact tables.

✅ **Enables Data Integration:**

- When data is collected from multiple sources, different systems may use different natural keys.
- A surrogate key provides a common identifier.

Example of Surrogate Key vs. Natural Key

Customer Dimension Table Without Surrogate Key (Using Natural Key)

Customer_ID	Name	City	Country
CUST_123	John	New York	USA
CUST_456	Emma	London	UK
CUST_789	Raj	Mumbai	India

✅ **Issue:**

- If Customer_ID changes in the source system (e.g., CUST_123 → CUST_999), the reference is broken.
- Different source systems may have different ID formats (123, CUST_123, CUST-XYZ).

Customer Dimension Table with Surrogate Key

Surrogate_Key	Customer_ID	Name	City	Country
1	CUST_123	John	New York	USA
2	CUST_456	Emma	London	UK
3	CUST_789	Raj	Mumbai	India

💡 **Surrogate Key (integer)** is used for joining with fact tables, making queries more efficient.

How Surrogate Keys Work in a Fact-Dimension Model

- **Fact tables** store only **surrogate keys** from dimension tables instead of business keys.
- This improves **performance and consistency**.

Example: Sales Fact Table with Surrogate Keys

Date_ID	Store_ID	Product_SK	Customer_SK	Sales_Amount
20240201	1	101	1	500
20240202	2	102	2	700

Generating Surrogate Keys

1. **Auto-Increment Columns (SQL Databases)**

```
CREATE TABLE Customer_Dim (
   Surrogate_Key INT IDENTITY(1,1) PRIMARY KEY,
   Customer_ID VARCHAR(50),
   Name VARCHAR(100),
   City VARCHAR(100),
   Country VARCHAR(100)
);
```

2. **Using UUIDs (For Distributed Systems)**

```
SELECT UUID() AS Surrogate_Key;
```

- Suitable for **distributed databases** (e.g., NoSQL, Hadoop).

3.**ETL Process (Incremental Assignment in Data Pipelines)**

- AWS Glue, Apache Spark, or Python scripts can **assign sequential surrogate keys** during ETL.

Surrogate Key vs. Natural Key: When to Use Each?

Feature	**Surrogate Key** ✅	**Natural Key** ❌
Uniqueness	Always unique	May change or be reused
Performance	Faster joins	Slower with long text keys
Tracking History (SCDs)	Easy (new key for changes)	Difficult
Integration Across Systems	Works well	Issues if different formats
Storage Size	Small (Integer)	Large (String/Text)

Real-World Example: Handling Slowly Changing Dimensions (SCD Type 2)

- Suppose **Emma (Customer_ID = CUST_456)** moves from **London → Paris**.
- Instead of updating the same row, a **new surrogate key** is created to maintain history.

Surrogate_Key	**Customer_ID**	**Name**	**City**	**Country**	**Start_Date**	**End_Date**
2	CUST_456	Emma	London	UK	2023-01-01	2024-02-10
4	CUST_456	Emma	Paris	France	2024-02-11	NULL

✅ **Now we can track customer history without overwriting past data.**

Conclusion

- **Surrogate keys are essential for data warehousing** due to their performance, uniqueness, and flexibility.
- They solve **natural key issues**, such as data changes and source system dependencies.
- Used **heavily in fact-dimension models** for **efficient joins, SCD handling, and data integration.**

Natural Keys in Data Warehousing & Dimensional Modelling

What is a Natural Key?

A **natural key** is a business-defined attribute (or set of attributes) that uniquely identifies a record in a database. Unlike surrogate keys (which are system-generated integers), natural keys come from the **source system** and have real-world meaning.

Examples of Natural Keys

Entity	Example Natural Key
Customer	Customer_ID (e.g., CUST_123)
Product	Product_Code (e.g., SKU_9876)
Employee	Employee_ID (e.g., EMP001)
Order	Order_Number (e.g., ORD_56789)
Vehicle	VIN (Vehicle Identification Number)

Advantages of Natural Keys

✅ **Business Meaningful**

- Natural keys represent **real-world entities** and are directly **understandable**.
- Example: A VIN uniquely identifies a vehicle, and a Social Security Number (SSN) uniquely identifies a person.

✅ **No Extra Storage Needed**

- Since natural keys **already exist** in source data, **no additional column** is needed for identification.
- Example: Instead of adding an artificial Surrogate_Key, we can use Employee_ID directly.

✅ **Maintains Business Integrity**

- Prevents duplicate records if properly enforced.
- Example: A duplicate Order_Number would indicate a system issue.

Disadvantages of Natural Keys

❌ **May Change Over Time**

- Some business keys are **not stable**.
- Example: A company **changes its customer ID format**, or an employee **gets a new ID after rehiring**.

❌ **Not Always Unique Across Systems**

- Different systems may use different key formats.
- Example: One system uses 12345, while another uses CUST-12345.

❌ **Performance Issues in Joins**

- Long string-based natural keys **increase storage size** and **slow down joins**.
- Example: Using Customer_Email as a key is inefficient in large datasets.

❌ **Complicated Composite Keys**

- Some tables require **multiple columns** to form a natural key, making queries complex.
- Example: A transaction table might need (Store_ID, Transaction_Date, Register_Number) as a key.

Natural Key vs. Surrogate Key: When to Use Each?

Feature	**Natural Key** ❌	**Surrogate Key** ✅
Uniqueness	May not always be unique	Always unique
Performance in Joins	Slower (text-based keys)	Faster (integer-based keys)
Stability Over Time	May change	Remains stable
Storage Size	Larger (strings, composite keys)	Smaller (integers)
Integration Across Systems	Difficult if formats vary	Easier

Example: Using Natural vs. Surrogate Keys in a Data Warehouse

Scenario: Customer Dimension Table

Without Surrogate Key (Using Natural Key)

Customer_ID	**Name**	**City**	**Country**
CUST_123	John	New York	USA
CUST_456	Emma	London	UK
CUST_789	Raj	Mumbai	India

🔴 **Issue:** If CUST_123 is renamed or reused, referential integrity is affected.

With Surrogate Key (Best Practice in Dimensional Modelling)

Surrogate_Key	Customer_ID	Name	City	Country
1	CUST_123	John	New York	USA
2	CUST_456	Emma	London	UK
3	CUST_789	Raj	Mumbai	India

✅ **Surrogate Key** ensures consistency and **prevents dependency** on business-defined keys.

When Should You Use a Natural Key?

✓ If the key is **stable and never changes** (e.g., VIN, Passport Number).
✓ If it is a **widely accepted identifier** (e.g., ISBN for books).
✓ If **query performance is not a major concern** (OLTP systems).

When Should You Avoid a Natural Key?

❌ If the key **changes frequently** (e.g., Customer ID format updates).
❌ If it is **long and complex**, affecting performance.
❌ If it is **not unique across multiple source systems**.

Conclusion

- **Natural keys** are **business-defined unique identifiers**, but they may change, be reused, or impact performance.
- **Surrogate keys** are **preferred in data warehousing** to ensure **consistency, stability, and faster joins**.
- A **hybrid approach** can be used: Keep the **natural key for reference** and **use a surrogate key for primary key relationships** in fact and dimension tables.

Dimension Tables in Dimensional Modelling

What is a Dimension Table?

A **dimension table** is a table in a data warehouse that stores **descriptive (textual) attributes** related to business entities. It provides **context** to the data stored in fact tables, enabling analysis and reporting.

Example Entities Stored in Dimension Tables:
✅ Customers
✅ Products
✅ Locations
✅ Time (Date)
✅ Employees

Each record in a dimension table represents a **single business entity** with descriptive details.

Key Characteristics of Dimension Tables

✅ **Stores Descriptive Data (Text-Based Attributes)**

- Example: A Customer dimension may contain **Customer Name, Address, Gender, and Segment**.

✅ **Contains a Primary Key (Surrogate Key)**

- Used to join with fact tables efficiently.
- Example: Customer_SK (Surrogate Key) links the Customer dimension with Sales Fact.

✅ **Denormalized for Performance**

- Unlike OLTP databases, dimension tables are **not highly normalized**.
- Reduces joins and speeds up queries.

✅ **Supports Hierarchies**

- Example: Date Dimension includes Year → Quarter → Month → Day.
- Example: Location Dimension has **Country → State → City**.

✅ **Slowly Changing Dimensions (SCDs) Are Managed**

- Historical changes (e.g., customer moves to a new city) are tracked using **SCD Type 2 (new record with a different surrogate key)**.

Example: Product Dimension Table

Product_SK	Product_ID	Product_Name	Category	Brand	Price	Launch_Date
101	P001	Laptop X1	Electronics	Dell	1000	2024-01-01
102	P002	Phone Z5	Electronics	Samsung	800	2023-10-15
103	P003	Smartwatch	Wearables	Apple	500	2023-06-20

💡 **Notes:**

- Product_SK (Surrogate Key) is used in **fact tables** for efficient joins.
- Product_ID is a **natural key** from the source system.
- The table contains **descriptive attributes** like Category and Brand.

Types of Dimension Tables in Dimensional Modelling

1. Conformed Dimension

- A **shared dimension** used across multiple fact tables and business processes.
- Ensures **data consistency** across different subject areas.
- Example: A **Customer Dimension** is used in both **Sales Fact** and **Support Fact** tables, ensuring that customer information remains consistent across different departments.
- Helps in creating an **enterprise-wide data warehouse** with standardized dimensions.

---

2. Role-Playing Dimension

- A **single dimension** used in different roles within the same fact table.
- The most common example is the **Date Dimension**, which can represent **Order Date, Shipment Date, and Payment Date** in a Sales Fact Table.
- Instead of duplicating the Date Dimension, we create **multiple aliases** while joining it to the fact table.
- Ensures **data consistency** while reducing redundancy in the model.

---

3. Slowly Changing Dimension (SCD)

- A **dimension that tracks changes** in attribute values over time.
- Common types:
    - **SCD Type 1 (Overwrite)**: Updates the value without keeping history (e.g., correcting a typo in a customer name).

    - **SCD Type 2 (Track History)**: Adds a **new row with a different surrogate key** for each change (e.g., if a customer moves to a new city, we keep both the old and new addresses).
    - **SCD Type 3 (Limited History)**: Stores the **previous and current values** in separate columns (e.g., keeping "Previous Address" and "Current Address" fields in the Customer Dimension).
- Used when business users require **historical tracking of dimension attributes**.

---

4. Junk Dimension

- A single table that stores **miscellaneous, low-cardinality attributes** (e.g., Flags, Yes/No values) that don't fit in other dimensions.
- Example: A **Customer Preference Dimension** storing fields like "Subscribed to Newsletter", "Loyalty Member", "Has Credit Card".
- Helps in **reducing the number of columns** in the fact table and avoids having too many small dimensions.
- **Optimizes storage and improves query performance** by grouping multiple binary/text attributes into a single dimension.

---

5. Degenerate Dimension (DD)

- A **dimension stored within the fact table itself** instead of having a separate table.
- Example: **Order Number, Invoice Number, or Transaction ID** in a Sales Fact Table.
- These attributes are **business keys** but **lack descriptive attributes**, so they don't need a separate dimension table.
- Useful when **querying at a transaction level** without needing additional joins.

---

6. Date Dimension (Time Dimension)

- A special dimension that stores **predefined time-related attributes** such as **Year, Quarter, Month, Week, Day, Holiday Flag, Fiscal Period, etc.**
- Instead of using a **DATE data type**, we use a **Surrogate Key (Date_SK)** to optimize queries and support different fiscal calendars.
- Example Structure:

Date_SK	Date	Day_of_Week	Month	Quarter	Year	Is_Holiday
20240201	2024-02-01	Thursday	Feb	Q1	2024	No
20240214	2024-02-14	Wednesday	Feb	Q1	2024	Yes (Valentine's Day)

- Helps in performing **date-based aggregations** efficiently.

- Used in almost **every data warehouse project**.

Conclusion

- **Conformed Dimensions** ensure **consistency across multiple fact tables**.
- **Role-Playing Dimensions** allow **reuse of the same dimension** in different contexts.
- **Slowly Changing Dimensions (SCDs)** help in **tracking historical changes** in attributes.
- **Junk Dimensions** group miscellaneous attributes to **optimize data storage**.
- **Degenerate Dimensions** store **transactional keys directly in fact tables**.
- **Date Dimension** is a critical table that enables **efficient time-based reporting**.

Nulls in Dimension Tables

Handling **NULL values** in dimension tables is crucial for ensuring data integrity, accurate reporting, and avoiding unexpected issues in joins with fact tables. Here's how NULLs impact dimension tables and how they should be managed:

Why Do NULLs Appear in Dimension Tables?

NULL values in dimension tables can occur due to:
✅ **Missing Data from Source Systems** – Example: A customer record without an email address.
✅ **Late Arriving Dimension Records** – A fact record arrives before its corresponding dimension record (e.g., an order is recorded before customer details are updated).
✅ **Data Entry Errors** – Incomplete records in transactional systems.
✅ **Unknown or Not Applicable Values** – Some attributes may not be relevant for certain records (e.g., a product with no brand name).

Problems Caused by NULLs in Dimension Tables

❌ **Issues in Joins with Fact Tables** – NULL values in foreign keys can lead to missing data when joining with fact tables.
❌ **Incorrect Aggregations in Reports** – NULLs can cause incorrect grouping in BI reports.
❌ **Complications in Slowly Changing Dimensions (SCDs)** – NULL values may create issues when tracking historical changes.

Strategies for Handling NULLs in Dimension Tables

1. Using Default Values Instead of NULLs

Instead of storing NULLs, **replace them with default values** to maintain referential integrity.

Customer_SK	Customer_Name	Email	City	Country
101	John Doe	johndoe@email.com	New York	USA
102	Jane Smith	(Unknown)	London	UK
103	Alex Brown	alex@email.com	(Unknown)	Canada

👉 **Example Replacements:**

- Missing email → Store "Unknown"
- Missing city → Store "Not Provided"

---

2. Using a Special "Unknown" or "Default" Record

If a dimension key is missing, insert a special record with a default **Surrogate Key (SK)**.

Example: Special Row in Customer Dimension

Customer_SK	Customer_Name	Email	City	Country
-1	Unknown	Unknown	Unknown	Unknown
101	John Doe	johndoe@email.com	New York	USA

👉 The fact table can reference this **"Unknown"** customer instead of having NULL foreign keys.

---

3. Handling Late Arriving Dimension Records

For **late arriving dimensions**, use a **dummy placeholder record** in the dimension table and update it when the actual data arrives.

🔹 **Step 1:** Insert a placeholder record (Customer_SK = -1).
🔹 **Step 2:** When the real customer data arrives, update it with correct details.

---

4. Using Surrogate Keys to Avoid NULL Foreign Keys

Instead of allowing NULL values in foreign key columns, **always ensure a valid surrogate key is assigned**.

- Instead of:

```
SELECT * FROM Sales_Fact sf

JOIN Customer_Dim cd ON sf.Customer_SK = cd.Customer_SK;
```

- Use:

SELECT * FROM Sales_Fact sf

JOIN Customer_Dim cd ON sf.Customer_SK = COALESCE(cd.Customer_SK, -1);

👉 This ensures missing customers map to a valid record (-1).

Conclusion

- NULLs in dimension tables can lead to **join issues and incorrect reports**.
- Use **default values** or an **"Unknown" surrogate key (-1)** to handle missing data.
- Implement **late arriving dimension handling** to update missing records.
- Always ensure **fact tables reference valid dimension keys** to maintain data integrity.

Hierarchies in Dimension Tables

A **hierarchy in dimensional modelling** represents **levels of data granularity** within a dimension, allowing users to analyse data at different levels of detail. Hierarchies help in **drill-down, roll-up, and summarization** operations in data warehouses and OLAP cubes.

🔷 **Types of Hierarchies in Dimensions**

1. Balanced Hierarchy (Level-Based Hierarchy)

- A **well-defined and structured hierarchy** where each level has a clear relationship with its parent level.
- All branches of the hierarchy **maintain the same number of levels**.
- Example: **Geographical Hierarchy**

Continent → Country → State → City

Table Example:

Continent	Country	State	City
North America	USA	California	Los Angeles
North America	USA	Texas	Houston
Asia	India	Karnataka	Bangalore

  - This hierarchy **allows aggregations** at Continent, Country, or State levels.
  - Queries can summarize data at any level, e.g., **total sales by continent or country**.

---

2. Unbalanced Hierarchy

- The **number of levels is not uniform** for all members.
- Example: **Organization Hierarchy (Employee Reporting Structure)**

CEO → VP → Manager → Employee

  - Some employees may report directly to the CEO, while others may have multiple levels of management.
  - The depth of hierarchy **varies across different branches**.

Table Example:

Employee_ID	Employee_Name	Manager_ID	Level
1	CEO	NULL	1
2	VP Sales	1	2
3	VP Finance	1	2
4	Sales Manager	2	3
5	Finance Analyst	3	3

  - Allows drill-down from **CEO → VP → Manager → Employee**.
  - Queries can retrieve **direct reports** for any employee dynamically.

---

3. Ragged Hierarchy

- Similar to **unbalanced hierarchies**, but some levels **may be skipped** entirely.
- Example: **Product Category Hierarchy**

Category → Subcategory → Product

  - Some products **may not have a subcategory**, going directly from **Category → Product**.

Table Example:

Category	Subcategory	Product
Electronics	Mobile Phones	iPhone 14
Electronics	NULL	MacBook Pro
Furniture	Chairs	Office Chair

  - The "MacBook Pro" **does not have a Subcategory** but still belongs to the hierarchy.

- Allows flexible analysis while **handling missing intermediate levels**.

---

4. Recursive Hierarchy (Parent-Child Hierarchy)

- A **self-referencing structure** where a record **refers to another record in the same table**.
- Example: **Employee-Manager Relationship**
  - Each employee reports to a **Manager_ID**, which is also an **Employee_ID** in the same table.

Table Example:

Employee_ID	Employee_Name	Manager_ID
1	CEO	NULL
2	VP Sales	1
3	VP Finance	1
4	Sales Manager	2
5	Sales Rep	4

SQL Query for Hierarchical Relationship

```
SELECT e1.Employee_Name AS Employee,
    e2.Employee_Name AS Manager
FROM Employee e1
LEFT JOIN Employee e2 ON e1.Manager_ID = e2.Employee_ID;
```

- Useful for **hierarchical reports** in HR or financial structures.

🔷 **Benefits of Hierarchies in Dimensional Modelling**

✅ **Efficient Aggregation** – Enables data summarization at multiple levels (e.g., sales at city, state, country levels).
✅ **Drill-Down & Roll-Up** – Users can drill down for details or roll up for summaries in BI reports.
✅ **Improved Query Performance** – Reduces the need for complex joins when querying data at different levels.
✅ **Better Data Organization** – Helps in structuring data for business intelligence and analytics.

Conclusion

Hierarchies are essential in dimensional modelling to support **multi-level reporting and analytics**. Depending on the business use case, different hierarchy types (**balanced, unbalanced, ragged, recursive**) are used to model relationships effectively.

Different Schemas in Dimensional Modelling

In **dimensional modelling**, schemas define how **fact** and **dimension** tables are structured and related within a data warehouse. The three most common schemas are:

1. **Star Schema**
2. **Snowflake Schema**
3. **Galaxy Schema (Fact Constellation Schema)**

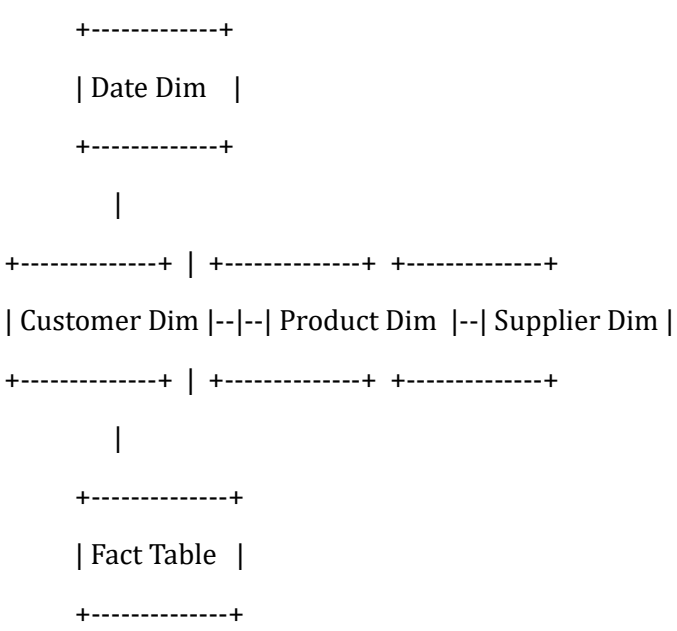
---

1. Star Schema ⭐

Definition

The **Star Schema** is the simplest and most widely used dimensional model, where a **central fact table** is directly connected to multiple **dimension tables**. The structure looks like a **star**, with the fact table at the center and dimension tables radiating outward.

Diagram Representation



Characteristics

✅ **Fact table** contains **measurable business data (facts)** with foreign keys referencing dimension tables.
✅ **Dimension tables** are **denormalized** (i.e., they store redundant data to improve query performance).
✅ Queries are **fast** because **fewer joins** are needed.

Example: Sales Data

Fact Table (Sales_Fact)

Sale_ID	Date_ID	Customer_ID	Product_ID	Amount	Quantity
1	20240201	101	201	500	2
2	20240202	102	202	300	1

Dimension Table (Customer_Dim)

Customer_ID	Customer_Name	City	Country
101	John Doe	NY	USA
102	Jane Smith	LA	USA

Advantages

✅ **Simple and easy to understand**
✅ **Fast query performance** (since dimension tables are denormalized)
✅ **Optimized for OLAP & BI reporting**

Disadvantages

❌ **Denormalization leads to redundancy** in dimension tables
❌ **Larger storage space is needed** compared to normalized schemas

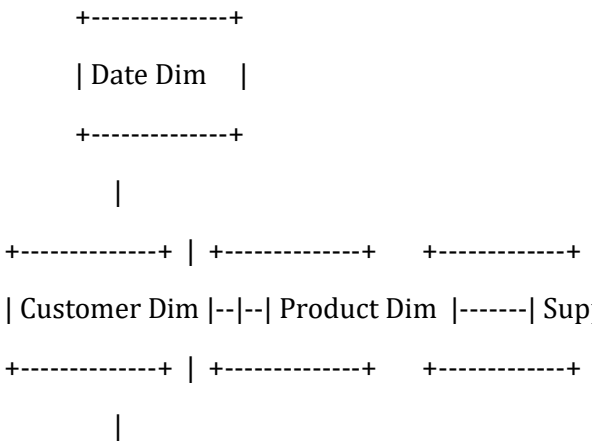
---

2. Snowflake Schema ❄️

Definition

The **Snowflake Schema** is an extension of the **Star Schema**, where **dimension tables are further normalized** into multiple related tables, reducing redundancy. The structure looks like a **snowflake** because dimension tables branch out.

Diagram Representation



```
+--------------+
| Fact Table   |
+--------------+
```

Characteristics

✅ **Fact table** remains the same as in **Star Schema**
✅ **Dimension tables are normalized** (split into sub-tables) to reduce data redundancy
✅ More **joins** are needed in queries

Example: Sales Data

Fact Table (Sales_Fact)

Sale_ID	Date_ID	Customer_ID	Product_ID	Amount	Quantity
1	20240201	101	201	500	2

Normalized Customer Dimension (Customer_Dim)

Customer_ID	Customer_Name	City_ID
101	John Doe	301
102	Jane Smith	302

City Dimension (City_Dim)

City_ID	City	Country
301	NY	USA
302	LA	USA

Advantages

✅ **Less storage space required** due to normalized dimension tables
✅ **Eliminates data redundancy**
✅ **Better data integrity**

Disadvantages

❌ **Query performance is slower** due to multiple joins
❌ **More complex structure** compared to Star Schema

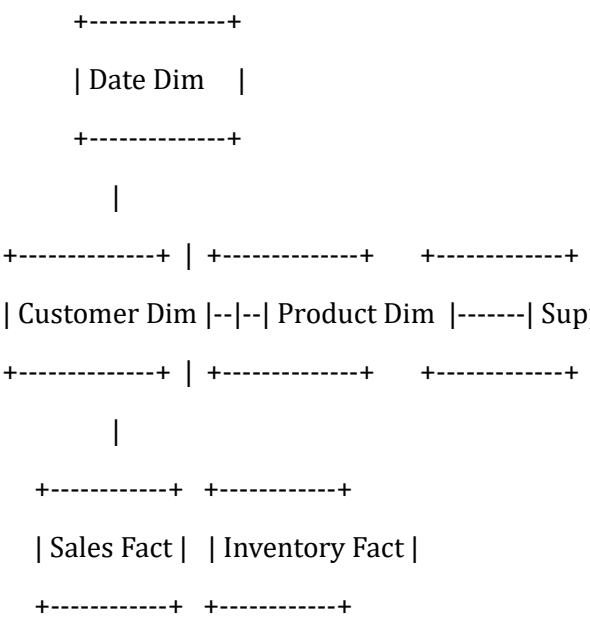
---

3. Galaxy Schema (Fact Constellation Schema) 🌌

Definition

The **Galaxy Schema** is a combination of **multiple fact tables** sharing common **dimension tables**. It is useful for complex business scenarios where multiple business processes are analysed together.

Diagram Representation



Characteristics

✅ **Multiple fact tables exist**, representing different business processes
✅ **Common dimension tables** are shared across fact tables
✅ Can represent **a large-scale enterprise data warehouse**

Example: Sales & Inventory Data

Fact Table 1: Sales_Fact

Sale_ID	Date_ID	Customer_ID	Product_ID	Amount	Quantity
1	20240201	101	201	500	2

Fact Table 2: Inventory_Fact

Inventory_ID	Date_ID	Product_ID	Stock_Level	Warehouse_ID
1	20240201	201	50	501

Advantages

✅ **Supports complex business use cases** (e.g., sales + inventory analysis)
✅ **Flexible and scalable** for large enterprises
✅ **Common dimensions help avoid data duplication**

Disadvantages

❌ **More complex schema design**
❌ **Query optimization is required for better performance**

🔷 Choosing the Right Schema

Feature	**Star Schema** ⭐	**Snowflake Schema** ❄️	**Galaxy Schema** 🌌
Complexity	Low	Medium	High
Query Performance	Fast	Slower due to joins	Medium (depends on query)
Storage Usage	High (denormalized)	Low (normalized)	High (multiple fact tables)
Use Case	Simple & Fast Reporting	Data Integrity & Storage Optimization	Large-scale DWs with multiple processes

---

🔷 Conclusion

- **Star Schema** → Best for **fast queries** and **simple reporting**.
- **Snowflake Schema** → Best for **storage efficiency** and **data integrity**.
- **Galaxy Schema** → Best for **complex data warehouses** with multiple fact tables.

Scenario: Designing a Data Model for a Telecommunications Company

Interviewer: "You're tasked with designing a data model for a telecommunications company... How would you approach this, considering factors like customer subscriptions, usage, billing, and customer support interactions?"

Approach for Designing a Data Model for a Telecommunications Company

Designing a **telecommunications data model** requires careful consideration of various business aspects, such as **customer subscriptions, usage, billing, and customer support interactions**. The goal is to create a scalable and efficient **dimensional model** that supports analytical and reporting needs.

Step 1: Understanding Business Requirements

Before starting the data modelling process, gather and analyse the business requirements by engaging with stakeholders. Key questions to consider:

🔷 **What are the business objectives?** (e.g., revenue tracking, customer retention analysis, fraud detection)
🔷 **Who are the primary users of the data?** (Finance, Operations, Customer Support, Marketing)
🔷 **What reports and KPIs need to be generated?** (e.g., Monthly Revenue, Churn Rate, Average Revenue per User - ARPU)
🔷 **What is the volume of data?** (e.g., daily call records, millions of customer transactions)
🔷 **What are the data sources?** (e.g., CRM, call logs, billing systems, support tickets)

Step 2: Identifying Key Business Entities and Relationships

To design an efficient model, identify the major business entities and their relationships. Key entities in a **telecom data model** typically include:

1. Customer Entity

- Represents the users of telecom services
- Attributes: Customer_ID, Name, Address, Contact_Info, Subscription_Status

2. Subscription Entity

- Tracks different service plans and packages
- Attributes: Subscription_ID, Customer_ID, Plan_Name, Activation_Date, Expiry_Date

3. Usage Entity

- Captures details of customer usage such as calls, messages, and data consumption
- Attributes: Usage_ID, Customer_ID, Subscription_ID, Call_Duration, Data_Used, SMS_Count, Timestamp

4. Billing Entity

- Stores customer invoices and payment history
- Attributes: Bill_ID, Customer_ID, Billing_Date, Amount, Payment_Status

5. Customer Support Interactions

- Logs customer complaints, queries, and resolutions
- Attributes: Ticket_ID, Customer_ID, Issue_Type, Resolution_Status, Assigned_Agent

---

Step 3: Choosing a Data Modelling Approach

Since this model is primarily for analytical/reporting purposes, we adopt **Dimensional Modelling** with **Fact and Dimension Tables**.

Step 4: Designing the Fact and Dimension Tables

A **dimensional model** consists of a **Fact Table** (contains measurable events) and multiple **Dimension Tables** (descriptive attributes).

✅ **Fact Tables (Transactional Data)**

Fact tables store **measurable business events** (e.g., usage, billing, support interactions).

Fact Table	Business Process Captured	Measures
Usage_Fact	Tracks call/data usage	Call Duration, Data Used, SMS Count
Billing_Fact	Tracks bill payments	Amount, Payment Status, Late Fee
Support_Fact	Tracks customer interactions	Response Time, Resolution Status

✅ **Dimension Tables (Descriptive Data)**

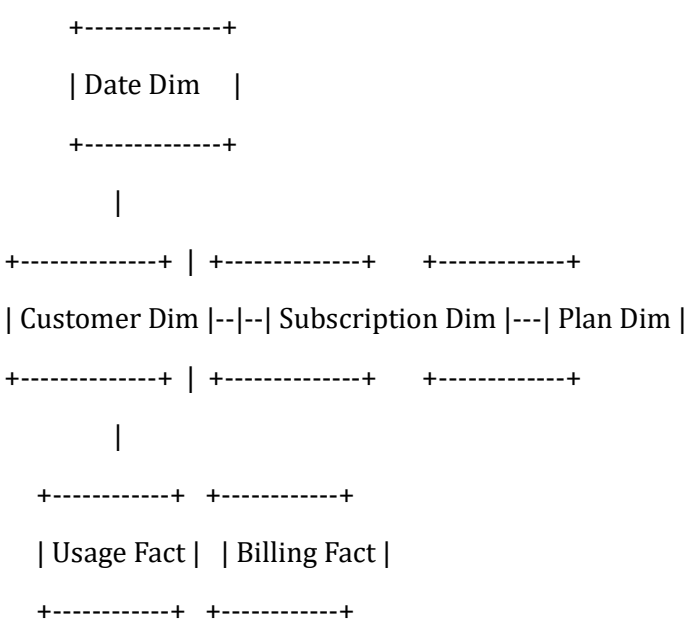
Dimension tables store descriptive attributes **linked to fact tables**.

Dimension Table	Attributes	Linked to Fact Table
Customer_Dim	Customer_ID, Name, Contact_Info	Usage_Fact, Billing_Fact, Support_Fact
Subscription_Dim	Subscription_ID, Plan_Name, Activation_Date	Usage_Fact
Date_Dim	Date_ID, Year, Month, Quarter, Day	All Fact Tables
Support_Agent_Dim	Agent_ID, Name, Department	Support_Fact

Step 5: Schema Selection

Based on the complexity, we choose:
✅ **Star Schema:** If we prioritize performance with fewer joins
✅ **Snowflake Schema:** If we need normalized dimension tables to reduce redundancy

📌 **Star Schema Example**



Step 6: Handling Data Integrity and Performance Optimization

🔷 **Surrogate Keys**

- Use **Surrogate Keys** (Auto-incremented) instead of **Natural Keys** (Customer Phone Number) to optimize joins.

🔷 **Partitioning & Indexing**

- Partition **Usage_Fact** on **Date_ID** for performance
- Index **Customer_ID** for faster lookup

🔷 **Handling Null Values**

- Default values for missing dimensions (e.g., "Unknown" in Customer Support)

Step 7: Data Pipeline Considerations

After modelling, the data warehouse must be populated using an **ETL pipeline** (Extract, Transform, Load).

- **Extract:** Pull data from CRM, Billing Systems, Call Logs
- **Transform:** Clean, deduplicate, standardize formats
- **Load:** Store data in **fact and dimension tables**

Step 8: Reporting & Business Insights

Once data is available in the model, build dashboards for business insights.

📊 **Examples of Reports:**

- **Customer Churn Analysis** – Identify customers at risk of leaving
- **Revenue Trends** – Track ARPU (Average Revenue Per User)
- **Support Performance Metrics** – Average resolution time
