UNIT II

Data Warehouse Design

Content

Data Modelling

Data modeling is the systematic process of **structuring and organizing** data in a data warehouse to support business intelligence (BI) and analytical requirements.

It defines how data is stored, processed, and retrieved efficiently, ensuring that data is consistent, high-quality, and optimized for query performance.

Key Objectives of Data Modelling

- To create a **blueprint** for how data is structured in a data warehouse
- To facilitate **efficient data retrieval** for decision-making and reporting
- To maintain **data consistency, integrity, and accuracy**
- To define **relationships** between different types of data to support analytical processing

Importance of Data Modelling in Data Warehouses

1. Structured Framework for Data Storage

- A well-designed data model provides a clear **structure** for how data is stored and organized.
- It ensures that data is logically grouped and can be retrieved efficiently.
- Reduces **redundancy** and improves **data management** by defining relationships between different entities.

2. Improves Query Performance and Analytics

- A properly modeled data warehouse ensures that queries run faster and more efficiently.
- Optimized **schema designs** (e.g., Star Schema, Snowflake Schema) improve **query execution times** by reducing the number of joins and redundant data.
- Well-defined indexing and partitioning strategies further enhance performance.

Importance of Data Modelling in Data Warehouses

3. Ensures Data Consistency and Integrity

- Data modeling ensures **data quality** by enforcing rules like **referential integrity** (i.e., foreign keys correctly linking tables).
- By normalizing data (in schemas like the **Snowflake Schema**), inconsistencies and anomalies are reduced.
- Data validation constraints ensure that only accurate and complete data enters the system.

4. Facilitates Efficient Data Retrieval for Reporting

- Business intelligence tools rely on well-modeled data for accurate reporting and analysis.
- Data models ensure that reports and dashboards provide the **right insights** in a timely manner.
- Enables **OLAP (Online Analytical Processing)** for slicing, dicing, and drilling down into the data.

Key Components of Data Modelling

Entities

Attributes

Relationships

Keys

Entities

An entity is any object, person, or concept about which data is collected and stored in a database. Entities represent real-world objects in the business domain.

Examples:

Customer: Represents individuals who make purchases

Product: Represents items sold in a business

Order: Represents transactions between customers and the business

Types of Entities:

Strong Entities: Exist independently and have a **primary key** (e.g., Customers, Products).

Weak Entities: Depend on another entity and **do not have a primary key** (e.g., Order Items in an invoice).

Attributes

Attributes are the characteristics or properties of an entity. They define the specific details stored about an entity.

Examples:

For a **Customer** entity, attributes could be:

- **Customer_ID** (Unique identifier)
- **Customer_Name** (Text)
- **Email** (Contact Information)
- **Location** (City, State, Country)

For a **Product** entity, attributes could be:

- **Product_ID** (Unique identifier)
- **Product_Name** (Text)
- **Category** (Electronics, Furniture, etc.)
- **Price** (Numeric value)

Types of Attributes:

Simple Attributes: Atomic values that cannot be divided further (e.g., Age, Name).

Composite Attributes: Attributes that can be split into sub-parts (e.g., Address → Street, City, State).

Derived Attributes: Attributes that are calculated from other attributes (e.g., Age derived from Date of Birth).

Multi-Valued Attributes: Attributes that can have multiple values (e.g., a person having multiple phone numbers).

Relationships

Relationships define the associations between different entities in a data model. They help connect data across multiple tables.

Examples:

A Customer places an Order (Customer → Order)

A Product belongs to a Category (Product → Category)

An Employee works in a Department (Employee → Department)

Types of Relationships:

One-to-One (1:1):

- One entity is related to only one instance of another entity.
- Example: Each **passport** is assigned to only one **person**.

One-to-Many (1:M):

- One entity is related to multiple instances of another entity.
- Example: **One Customer** can place **multiple Orders**.

Many-to-Many (M:N):

- Multiple instances of one entity relate to multiple instances of another entity.
- Example: **Students enroll in multiple Courses, and Courses have multiple Students**.
- This relationship usually requires a **junction table** (e.g., Enrollment Table).

Keys

Keys are **unique identifiers** used to define records within a table and establish relationships between tables.

Types of Keys:

- **Primary Key (PK):**
  - A **unique identifier** for each record in a table.
  - Example: Customer_ID in the **Customer table** uniquely identifies each customer.
- **Foreign Key (FK):**
  - A key in one table that refers to the **Primary Key of another table**.
  - Example: Customer_ID in the **Orders table** is a foreign key linking to the Customers table.
- **Candidate Key:**
  - A set of attributes that could be a primary key but only one is chosen.
  - Example: In a **Student table**, Student_ID and Email are candidate keys, but only one is chosen as the Primary Key.
- **Composite Key:**
  - A key made up of **two or more columns** to uniquely identify a record.
  - Example: Order_ID + Product_ID in an **Order Details table**.
- **Surrogate Key:**
  - A system-generated unique identifier, often a numeric or UUID value.
  - Example: A **database-generated sequence number** used instead of Customer_ID.

Real-World Case Studies and Examples of Data Models in Businesses

- Case Study 1: Data Modeling for an E-Commerce Business
- **Business Scenario:**
  - An online retail company, **ShopEasy**, wants to design a data warehouse to store transactional data and support business intelligence (BI) for analyzing customer purchasing behavior, product trends, and sales performance.
- Entities and Attributes:

Entity	**Attributes**
Customer	Customer_ID (PK), Name, Email, Location, Phone_Number
Product	Product_ID (PK), Name, Category, Price, Stock_Quantity
Order	Order_ID (PK), Customer_ID (FK), Order_Date, Total_Amount
Order_Details	Order_ID (FK), Product_ID (FK), Quantity, Subtotal
Category	Category_ID (PK), Category_Name

Real-World Case Studies and Examples of Data Models in Businesses

- **Relationships**:
  1. **One-to-Many (1:M):** A **Customer** can place multiple **Orders**.
  2. **Many-to-Many (M:N):** An **Order** contains multiple **Products**, and a **Product** can be in multiple **Orders**. This is resolved using the **Order_Details** table.
  3. **One-to-Many (1:M):** A **Product** belongs to a single **Category**, but a **Category** can have multiple **Products**.
- **Schema Used: Star Schema**
  - **Fact Table:** Orders (stores transaction details like sales and revenue).
  - **Dimension Tables:** Customers, Products, Categories, Time (for tracking daily sales).
- **Benefits of This Model:**
  - ✅ **Faster query execution** for sales analysis
    ✅ **Efficient reporting** on customer purchase behavior
    ✅ **Simplified schema structure** using star schema

Case Study 2: Data Modelling for a Healthcare Organization

- **Business Scenario:**
- A hospital management system needs a data warehouse to analyse patient treatment history, doctor performance, and hospital resource utilization.
- Entities and Attributes:

Entity	Attributes
Patient	Patient_ID (PK), Name, DOB, Gender, Contact, Insurance_Details
Doctor	Doctor_ID (PK), Name, Specialization, Experience
Appointment	Appointment_ID (PK), Patient_ID (FK), Doctor_ID (FK), Appointment_Date, Diagnosis
Treatment	Treatment_ID (PK), Patient_ID (FK), Doctor_ID (FK), Medication, Treatment_Date
Hospital_Department	Department_ID (PK), Name, Location

Case Study 2: Data Modeling for a Healthcare Organization

Relationships:

1. **One-to-Many (1:M):** A **Doctor** can have multiple **Appointments**.
2. **One-to-Many (1:M):** A **Patient** can have multiple **Treatments** over time.
3. **One-to-Many (1:M):** A **Hospital_Department** can have multiple **Doctors**.

- **Schema Used: Snowflake Schema**
  - The **Doctor** table is normalized (Specialization stored separately).
  - The **Patient** table is normalized (Insurance details in another table).
- **Benefits of This Model:**
  - ✅ **Reduces data redundancy** and ensures data integrity
    ✅ **Supports complex queries efficiently**
    ✅ **Allows better analysis of patient history and hospital performance**

FROM REQUIREMENTS TO DATA DESIGN

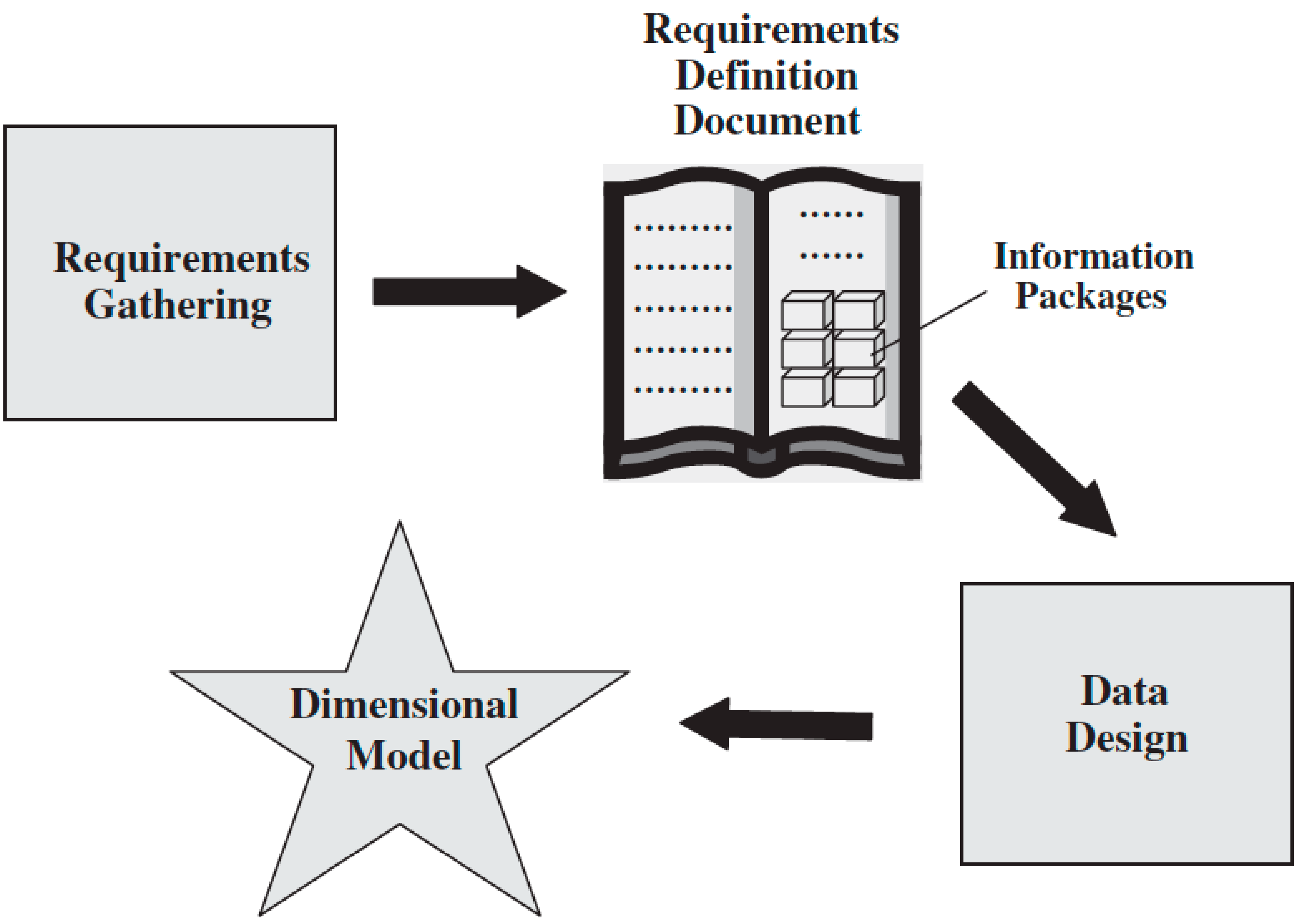


Figure 10-1 From requirements to data design.

- This process is **iterative** in many organizations:
  - feedback loops often go back to refine requirements,
  - adjust information packages, or
  - modify the design based on new insights or changing business needs.
- this figure highlights the **flow from high-level business requirements to a structured, analysis-ready data design** that meets those requirements.

FROM REQUIREMENTS TO DATA DESIGN

Start with Requirements Gathering: Identify what the business needs in terms of reports, dashboards, and analytics.

Create the Requirements Definition Document: Consolidate and clarify the needs, aligning technical scope with business priorities.

Develop Information Packages: Break down the needs into facts (metrics) and dimensions (descriptive attributes), defining the level of detail.

Proceed to Data Design: Use the information packages to model your data warehouse schema (e.g., star schema), plan your ETL processes, and address performance considerations.

Dimensional Model: Finalize the star or snowflake schema that will be implemented in the data warehouse, ensuring it supports all identified requirements.

Star schema

- A **star schema** places one **fact table** at the centre, surrounded by multiple **dimension tables**.
- Each dimension table has a **direct relationship** to the fact table, typically a one-to-many relationship (one dimension record → many fact records).
- This design simplifies queries: analysts can easily join the fact table to any dimension table to filter or aggregate data.
- **Star Schemas Are Effective**
  - **Query-Centric:** They are optimized for **read** operations (analysis and reporting).
  - **Simple Joins:** Because all dimension tables connect directly to the fact table, queries involve fewer, more straightforward joins.
  - **Equal Participation:** Each dimension can be used independently in queries to slice and dice the metrics.

A Simple Star Schema Example: Order Analysis

1. **Fact Table: Orders_Fact**
    1. In the centre of the schema.
    2. Holds metrics like **order dollars**, **cost**, **profit margin**, **quantity sold**.
2. **Dimension Tables** (Surrounding the fact table):
    1. **Customer_Dimension**
    2. **Salesperson_Dimension**
    3. **OrderDate_Dimension**
    4. **Product_Dimension**
3. **Sample Queries**
    1. “What was the **quantity sold** and **order dollars** for product bigpart-1 to customers in **Maine**, sold by salesperson **Jane Doe**, during **June**?”
    2. The fact table metrics (quantity, order dollars) are **filtered** by attributes in each dimension table (product name, state, salesperson name, month).
4. **Drilling Down and Rolling Up**
    1. A common requirement is to **start at summary** levels and then move to **greater detail**.
    2. For example, from **year 2008** (aggregated) → break down by **quarters** → then by **individual products** → finally by **states** in a region.
    3. The star schema’s dimension tables, which contain hierarchical attributes, make this **straightforward**.

Querying the Star Schema

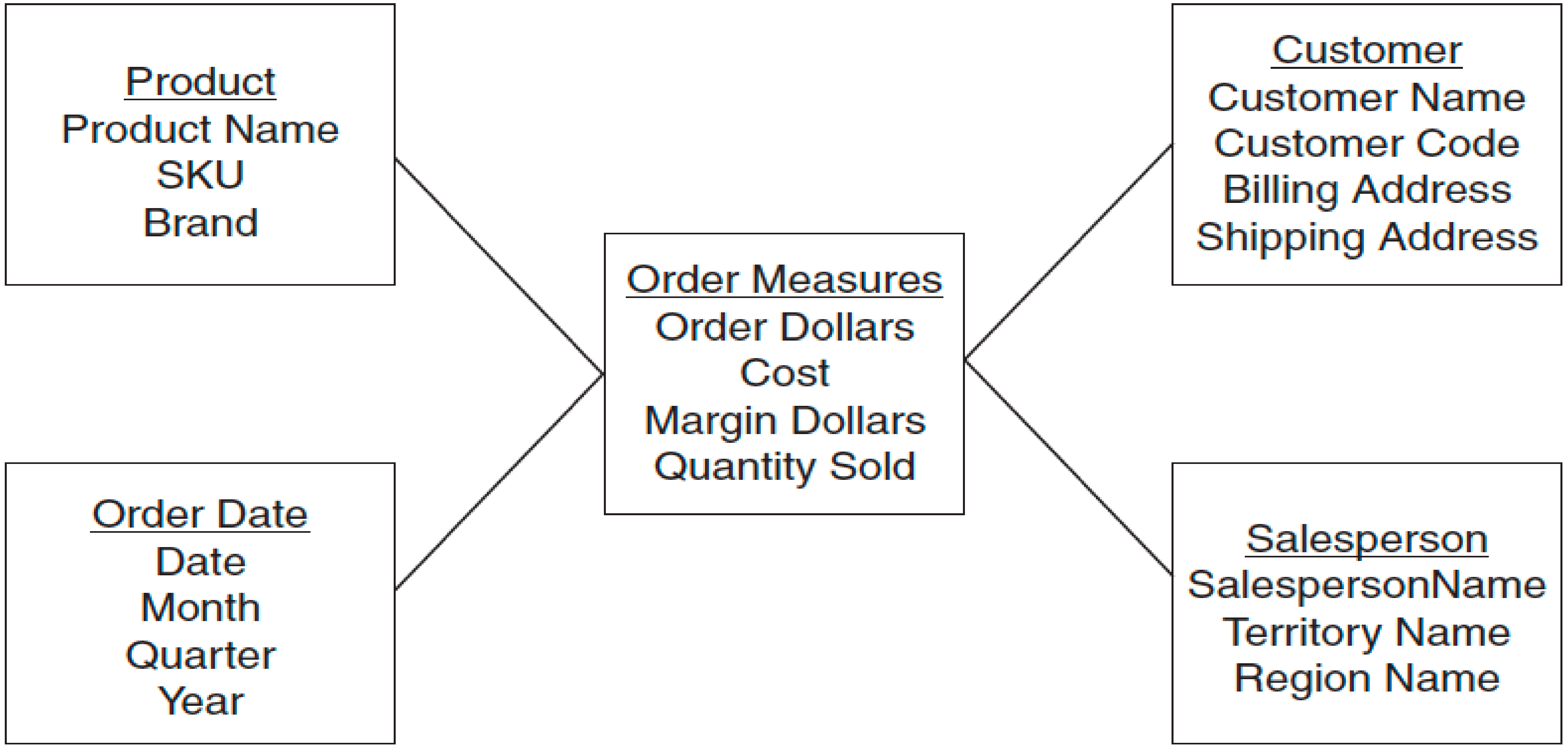
- **Join Patterns**
  - Queries join the **fact table** with **one or more dimension tables**.
  - Example: SELECT SUM(order_dollars) FROM Orders_Fact f JOIN Product_Dimension p ON f.product_key = p.product_key WHERE p.product_name = 'bigpart-1';
- **Constraints and Filters**
  - Attributes in dimension tables (e.g., state, month, salesperson_name) serve as **filters** or **grouping** criteria.
  - Users can easily see which attributes are available by **reviewing the star schema** diagram.
- **User Intuition**
  - Because the schema mirrors how users think about “who,” “what,” “when,” and “where,” it is **user-friendly** for writing ad-hoc queries.

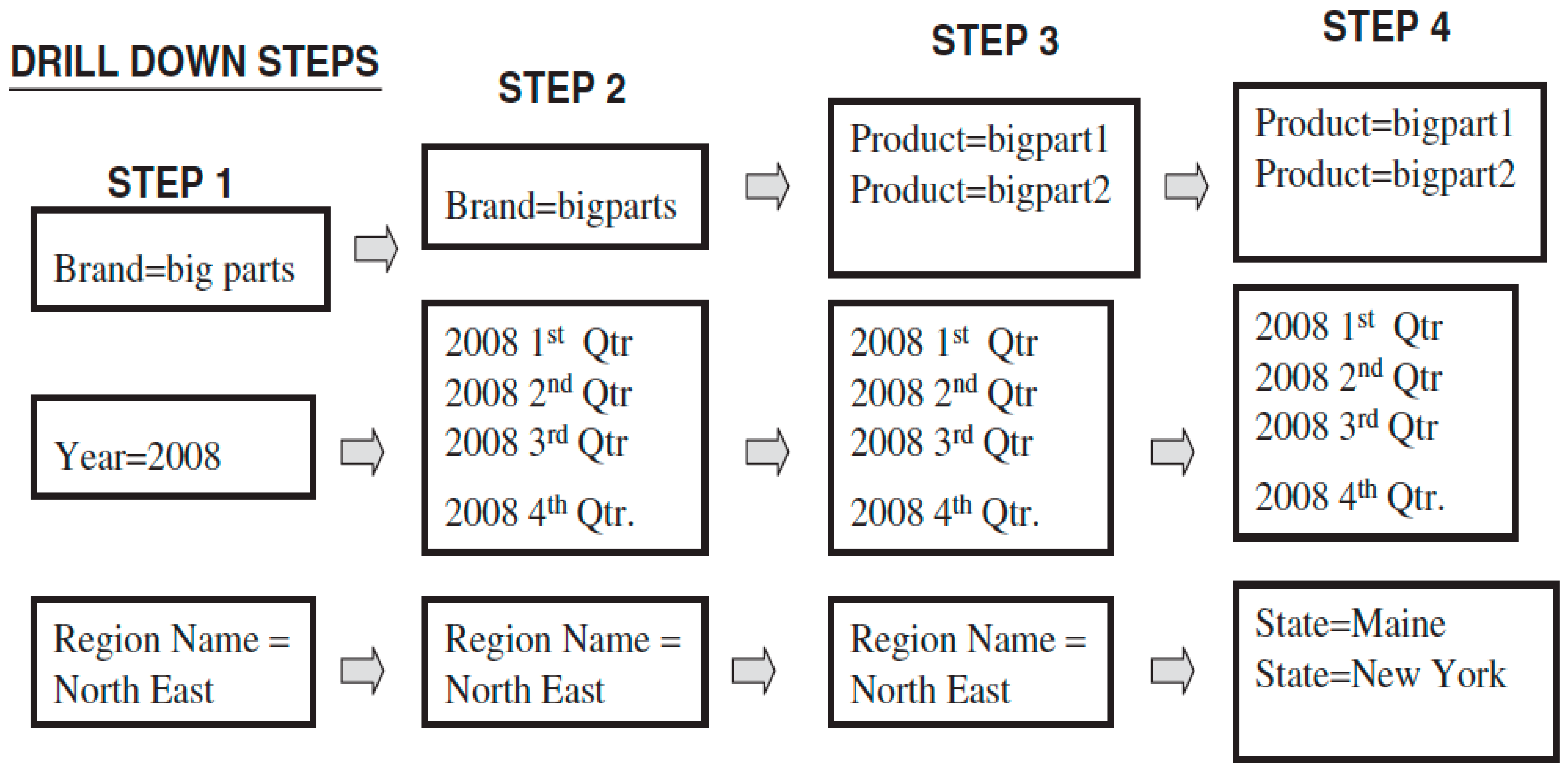
Inside a Dimension Table

1. **Dimension Table Key**
    1. Each dimension table has a **primary key** (often a **surrogate key**) that uniquely identifies each row.
    2. This key is used as a **foreign key** in the fact table.
2. **Table Is "Wide"**
    1. Dimension tables typically contain **many attributes** (sometimes 50+), making them **horizontally wide**.
3. **Textual Attributes**
    1. Attributes in dimension tables are usually **descriptive** or **categorical** (e.g., product category, city, region).
    2. Rarely contain **numeric fields** used for arithmetic calculations (those are typically in the fact table).
4. **Attributes Not Always Interrelated**
    1. Many attributes simply describe the dimension from different angles (e.g., package_size vs. brand in a product dimension).
    2. They need not be related to each other; they just provide various descriptive elements.

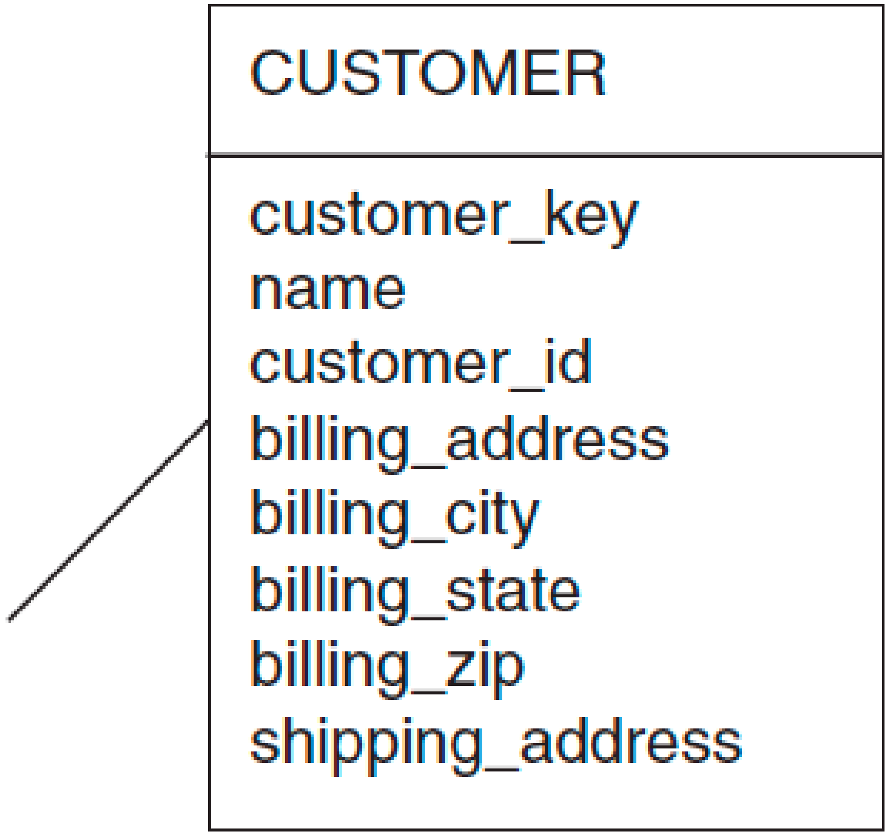
Inside a Dimension Table

1. **Not Normalized**
   1. Dimension tables are often **denormalized** to avoid extra joins, improving query performance.
   2. Flattening out attributes into a single table is common in star schemas.
2. **Drilling Down and Rolling Up**
   1. Dimension tables may contain **hierarchies** (e.g., state → city → zip code).
   2. Users can “drill down” to more detailed levels or “roll up” to more aggregated levels.
3. **Multiple Hierarchies**
   1. A single dimension can support **multiple hierarchies** (e.g., product classification for marketing vs. accounting).
   2. Each hierarchy is just another set of attributes.
4. **Fewer Records**
   1. Dimension tables generally have **far fewer rows** than fact tables.
   2. Example: A product dimension for an automaker might have 500 rows, while the fact table has millions.


Product
Product Name
SKU
Brand
Order Date
Date
Month
Quarter
Year
Order Measures
Order Dollars
Cost
Margin Dollars
Quantity Sold
Customer
Customer Name
Customer Code
Billing Address
Shipping Address
Salesperson
SalespersonName
Territory Name
Region Name


DRILL DOWN STEPS
STEP 1
Brand=big parts
Year=2008
Region Name = North East
STEP 2
Brand=bigparts
2008 1st Qtr
2008 2nd Qtr
2008 3rd Qtr
2008 4th Qtr.
Region Name = North East
STEP 3
Product=bigpart1
Product=bigpart2
2008 1st Qtr
2008 2nd Qtr
2008 3rd Qtr
2008 4th Qtr.
Region Name = North East
STEP 4
Product=bigpart1
Product=bigpart2
2008 1st Qtr
2008 2nd Qtr
2008 3rd Qtr
2008 4th Qtr.
State=Maine
State=New York

- Dimension table key
- Large number of attributes (wide)
- Textual attributes
- Attributes not directly related
- Flattened out, not normalized
- Ability to drill down/roll up
- Multiple hierarchies
- Less number of records



Inside a Fact Table

1. **Concatenated Primary Key**
    1. A fact table’s primary key is often a **composite** of the **foreign keys** referencing all related dimension tables.
    2. For example, Orders_Fact might have (product_key, order_date_key, customer_key, salesperson_key) as its concatenated primary key.
2. **Data Grain (Level of Detail)**
    1. **Granularity** indicates the **lowest level** of detail in the fact table (e.g., per day vs. per transaction).
    2. Determining the grain is crucial: it impacts **query flexibility**, **storage size**, and **maintenance**.
3. **Fully Additive Measures**
    1. Values that can be **summed** across dimensions (e.g., order_dollars, quantity_ordered).
    2. Most common type of measure used in aggregations.
4. **Semi-Additive Measures**
    1. Metrics that **cannot** simply be summed across all dimensions (e.g., margin_percentage).
    2. Calculations must be handled carefully (e.g., computing averages instead of sums).

Inside a Fact Table

1. **Narrow Table, Many Rows**
   1. Fact tables usually have **fewer attributes** (e.g., 10 or fewer) but **millions (or billions) of rows**.
   2. Visually, they are “deep” but not wide.
2. **Sparsity of Data**
   1. Not every combination of dimension attributes will have a corresponding fact table row.
   2. Fact tables can have “gaps,” storing only rows where actual events (e.g., orders) occurred.
3. **Degenerate Dimensions**
   1. Some **operational attributes** (e.g., order_number, invoice_number) are **neither measures nor standard dimension attributes**.
   2. These may still appear in the fact table for **analysis** (e.g., to count items per order).
4. **Factless Fact Tables**
   1. Certain **event-tracking** scenarios do not need numeric facts; the presence of a row indicates an event (e.g., student attendance).
   2. These “factless” tables still have a concatenated key referencing multiple dimensions but **no numeric measures**.

- Concatenated fact table key
- Grain or level of data identified
- Fully additive measures
- Semi-additive measures
- Large number of records
- Only a few attributes
- Sparsity of data
- Degenerate dimensions

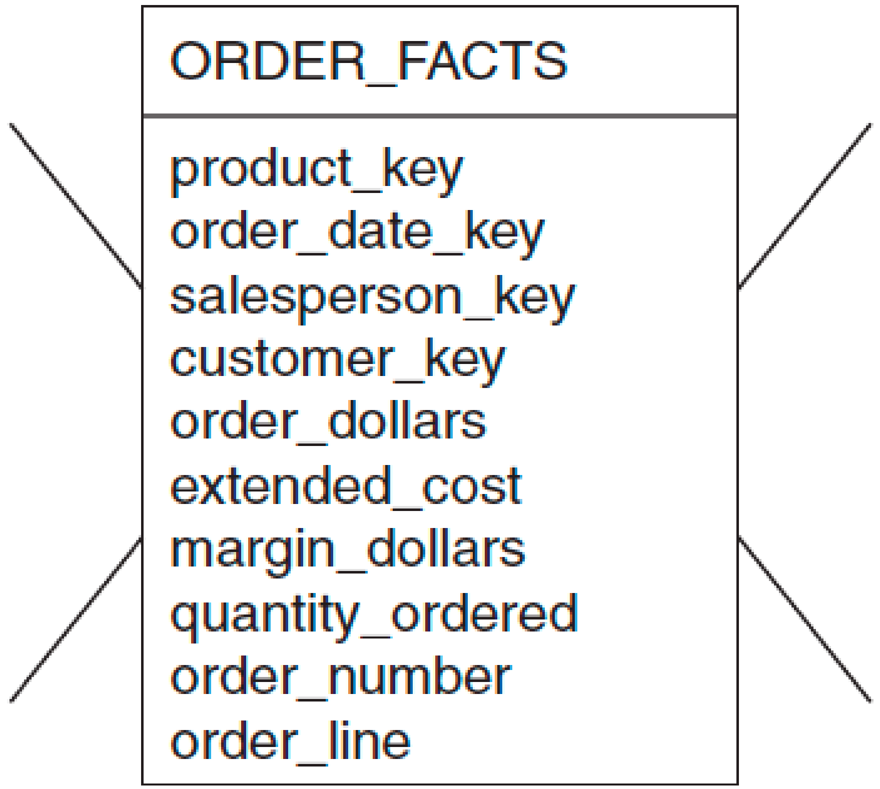


Figure 10-11 Inside a fact table.

Star Schema Keys

- **Primary Keys for Dimension Tables**
- **Natural Candidate Keys**
  - Examples:
    - **Product Dimension:** An 8-position product code (which might encode warehouse and category information).
    - **Customer Dimension:** A customer number.
    - **Sales Representative Dimension:** A social security number.

Star Schema Keys

Problems with Production System Keys

- When keys have embedded meanings (e.g., a product code indicating a warehouse), changes in operational circumstances (like moving a product to a different warehouse) force a change in the key.
- In some cases, customer numbers for discontinued customers might be reassigned, causing ambiguity and incorrect aggregation in historical data.

The Solution: Surrogate Keys

- Surrogate keys are system-generated sequence numbers that serve as primary keys for dimension tables.

Benefits:

- They have **no built-in meaning**, ensuring stability over time.
- The original operational system keys can be retained as non-key attributes for reference and auditing.

Example:

- In the Store dimension, a STORE_KEY (surrogate key) is used as the primary key, while the operational system's store code is stored in another column.

Foreign Keys in the Fact Table

Relationship Between Fact and Dimension Tables

- Each dimension table's primary key appears in the fact table as a **foreign key**.
- These keys link each fact row to its descriptive attributes in the dimensions (e.g., product, date, customer, salesperson).

Options for Fact Table Primary Keys

- **Option 1: Compound Primary Key**
  - Uses a single compound key formed by combining the keys from each dimension table, with additional duplicate storage of each foreign key.
  - **Drawback:** Increases fact table size.
- **Option 2: Concatenated Primary Key**
  - The primary key is the concatenation of all dimension table keys, and these keys serve directly as foreign keys.
  - **Advantage:** Most common in practice because it directly relates fact rows to dimension rows without redundancy.
- **Option 3: Generated Key**
  - Uses a system-generated key independent of dimension keys while still storing each foreign key as an attribute.
  - **Drawback:** Also increases table size.
- **Common Practice:**
  - **Option 2** is typically used because it simplifies the relationship and minimizes extra storage.

Advantages of the Star Schema

Simplified Querying

- **Direct Joins:** Since each dimension table is directly linked to the fact table, SQL queries are straightforward and often require fewer complex joins.
- **User-Friendly:** Business users and analysts can easily navigate the schema, even without deep technical knowledge.

Improved Query Performance

- **Denormalization:** Dimension tables are intentionally denormalized, which reduces the number of join operations required during query execution.
- **Optimized for Aggregation:** The structure is highly suitable for aggregation queries (summing sales, counting transactions, etc.), which are common in reporting and analysis.

Scalability and Maintenance

- **Modular Design:** Adding new dimensions or measures can be done with minimal impact on existing queries.
- **Data Warehousing Efficiency:** The star schema is optimized for read-heavy environments, making it an excellent choice for business intelligence and data analysis.

STAR SCHEMA: EXAMPLES

- Notice the following carefully and understand the STAR schema:
  - The metrics or facts being analyzed
  - The business dimensions used for analysis
  - The hierarchy available within each dimension for drill down and roll up
  - The primary and foreign keys

Video Rental

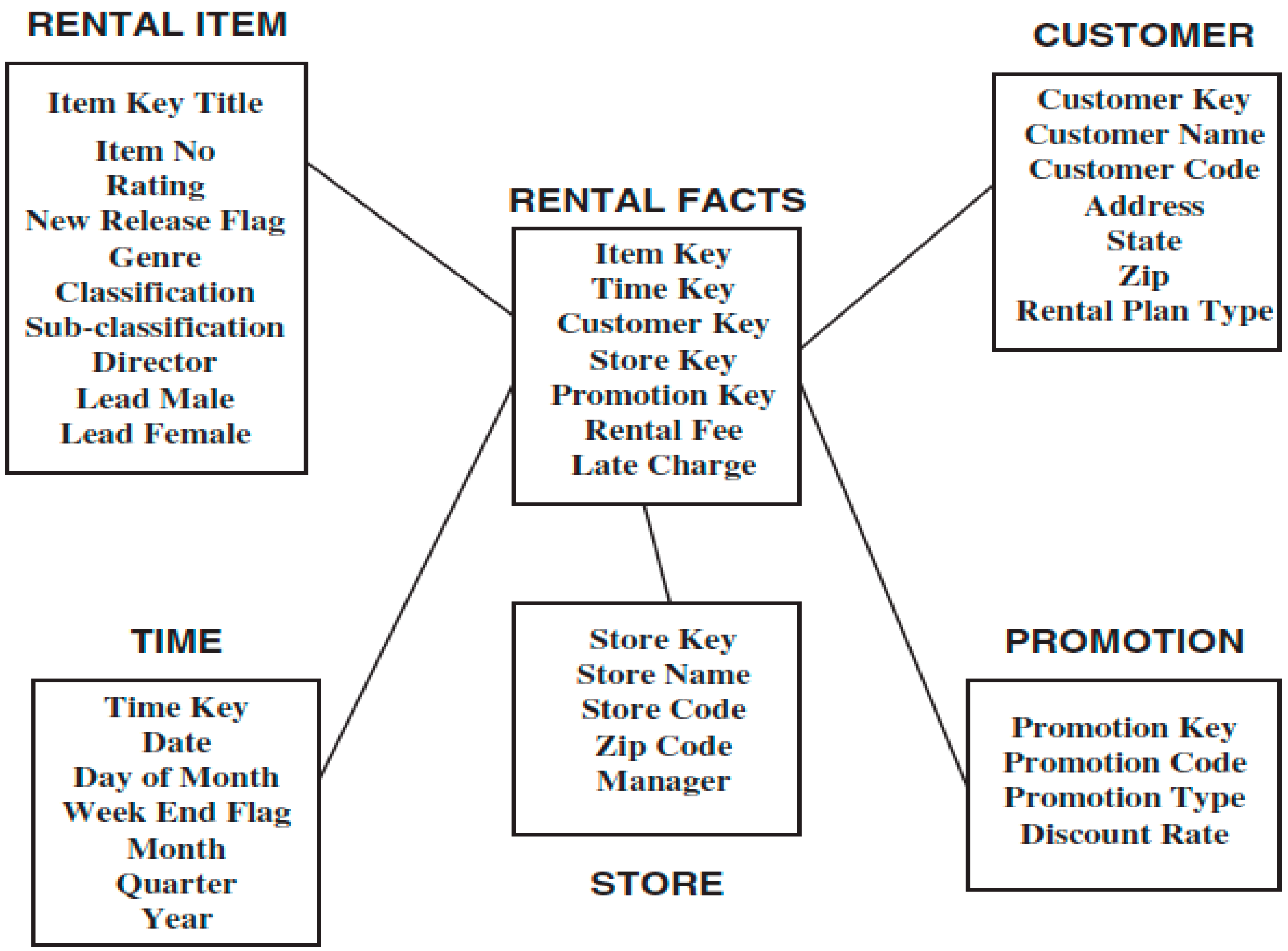


Figure 10-15 STAR schema example: video rental.

STAR schema example: supermarket.

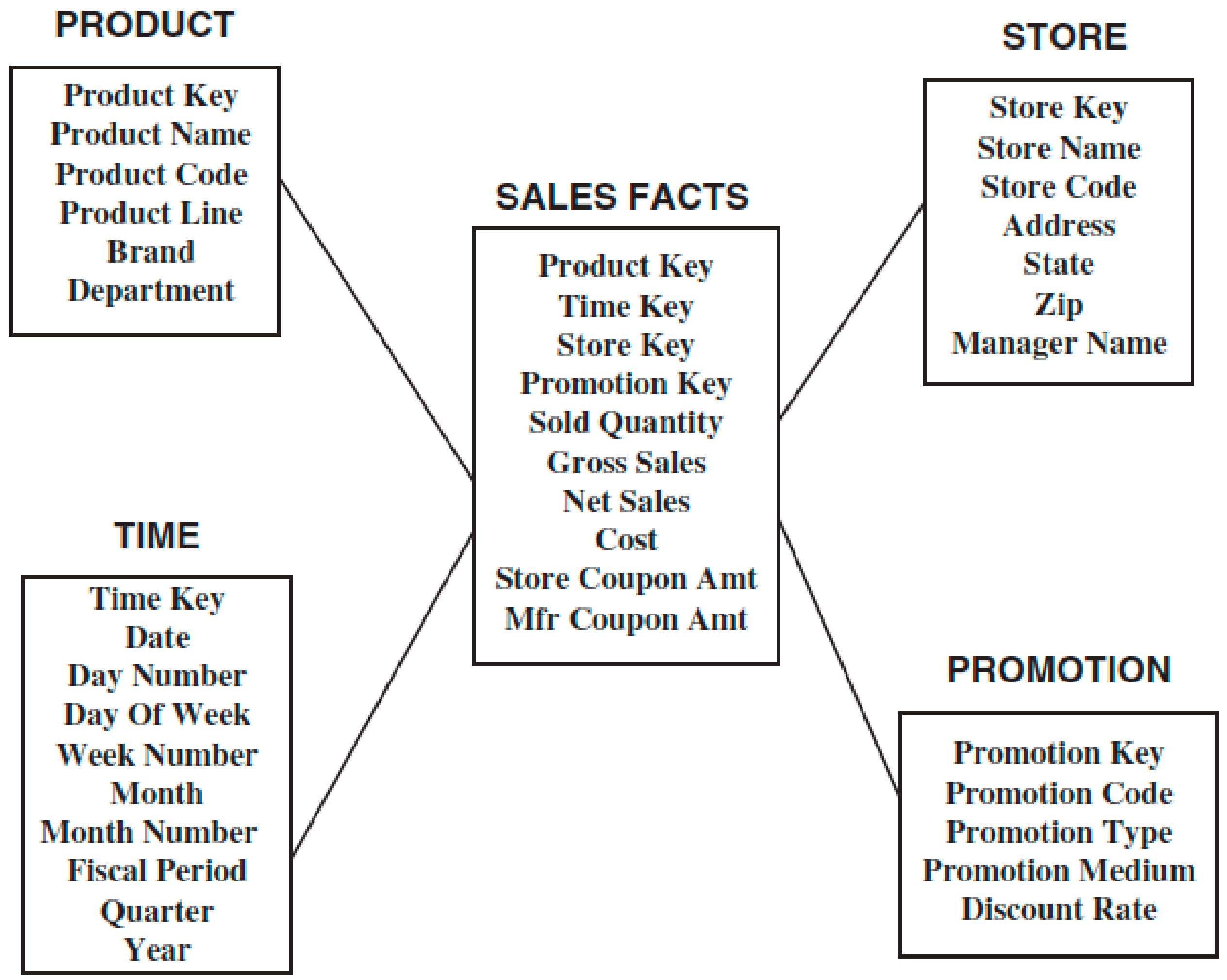


Figure 10-16 STAR schema example: supermarket.

STAR schema example: wireless phone service

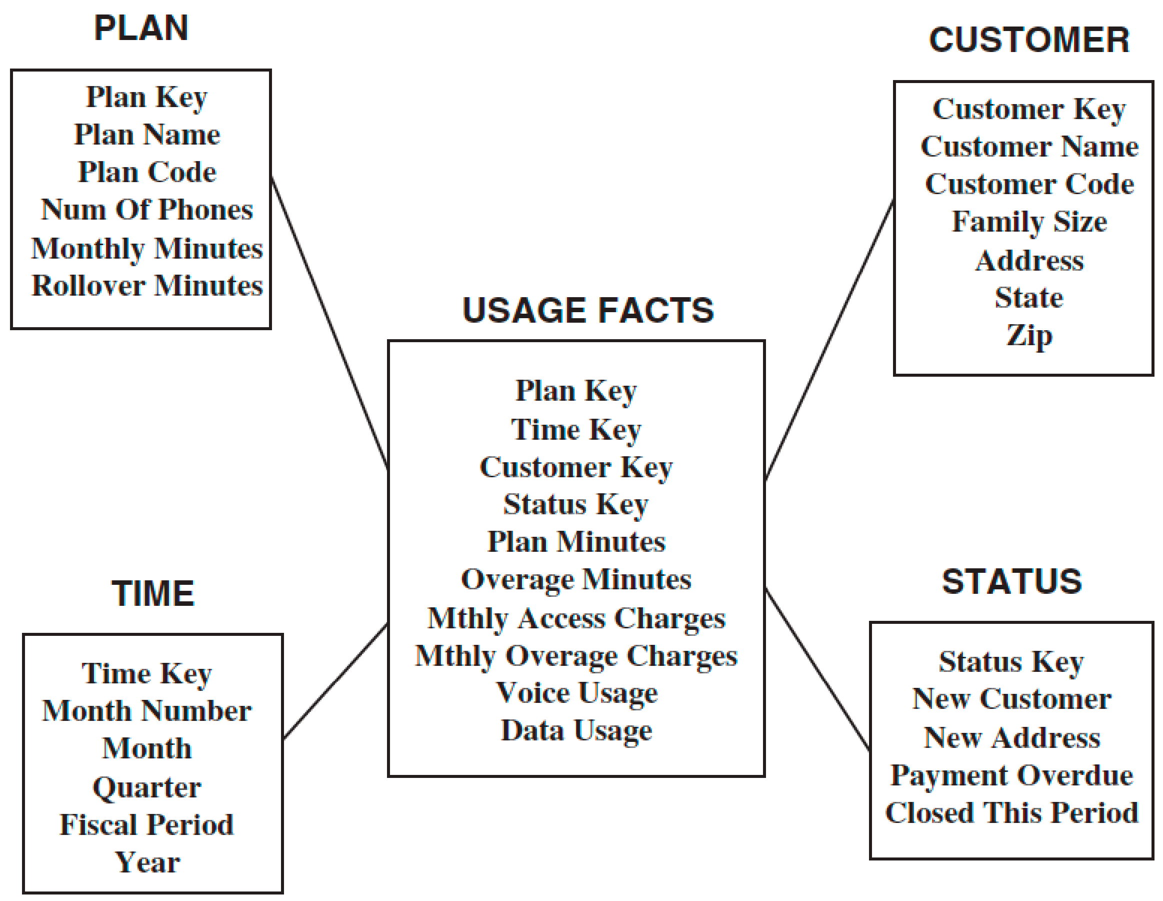


Figure 10-17 STAR schema example: wireless phone service.

STAR schema example: auction company

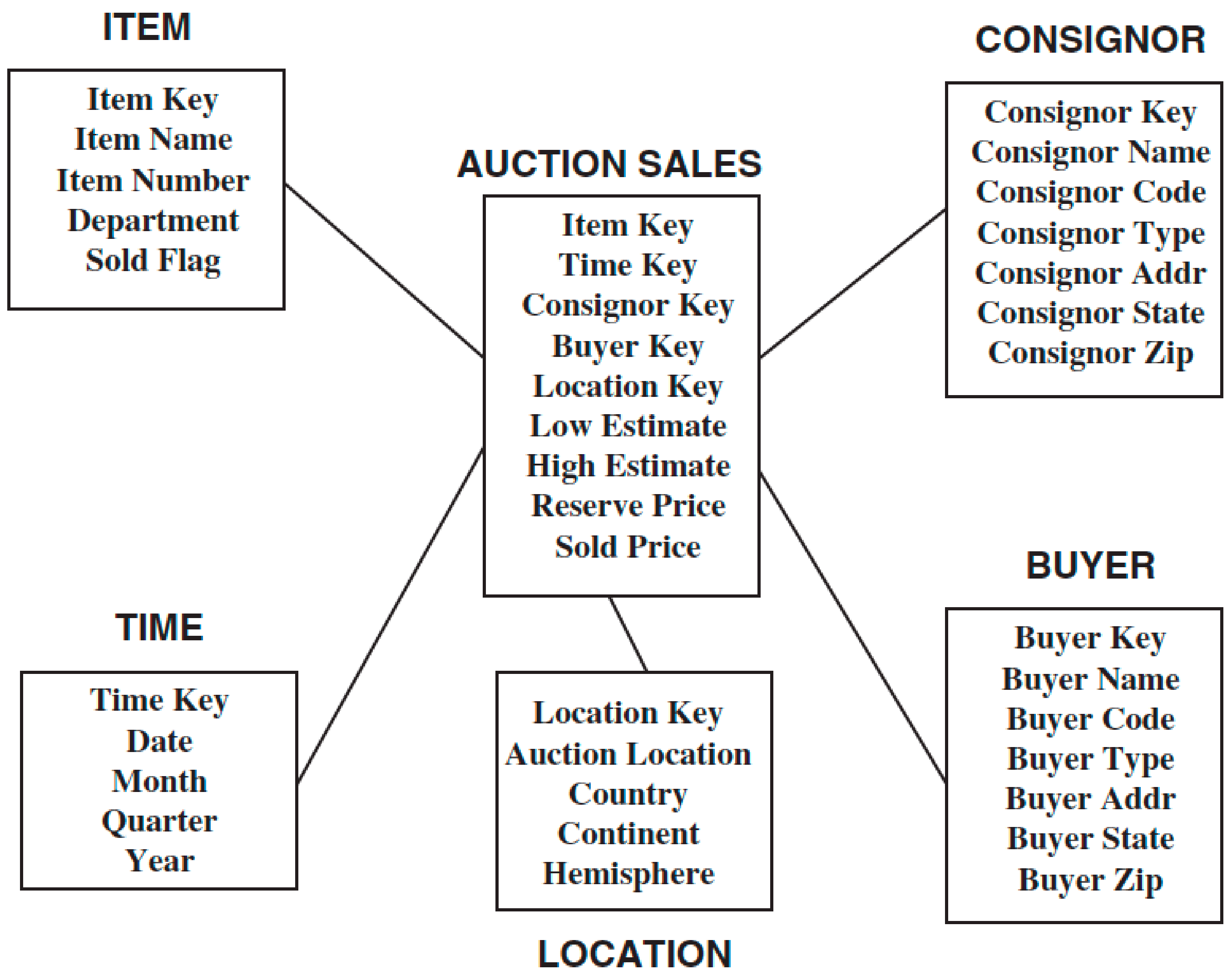


Figure 10-18 STAR schema example: auction company.

snowflake schema

The **Snowflake Schema** is a type of database schema commonly used in data warehousing.

It is an extension of the **Star Schema**, where the dimension tables are further normalized to reduce redundancy.

This results in a structure where dimension tables are split into multiple related tables, resembling a snowflake shape.

Key Characteristics of Snowflake Schema

Normalization: Dimension tables are divided into multiple related tables to remove redundancy.

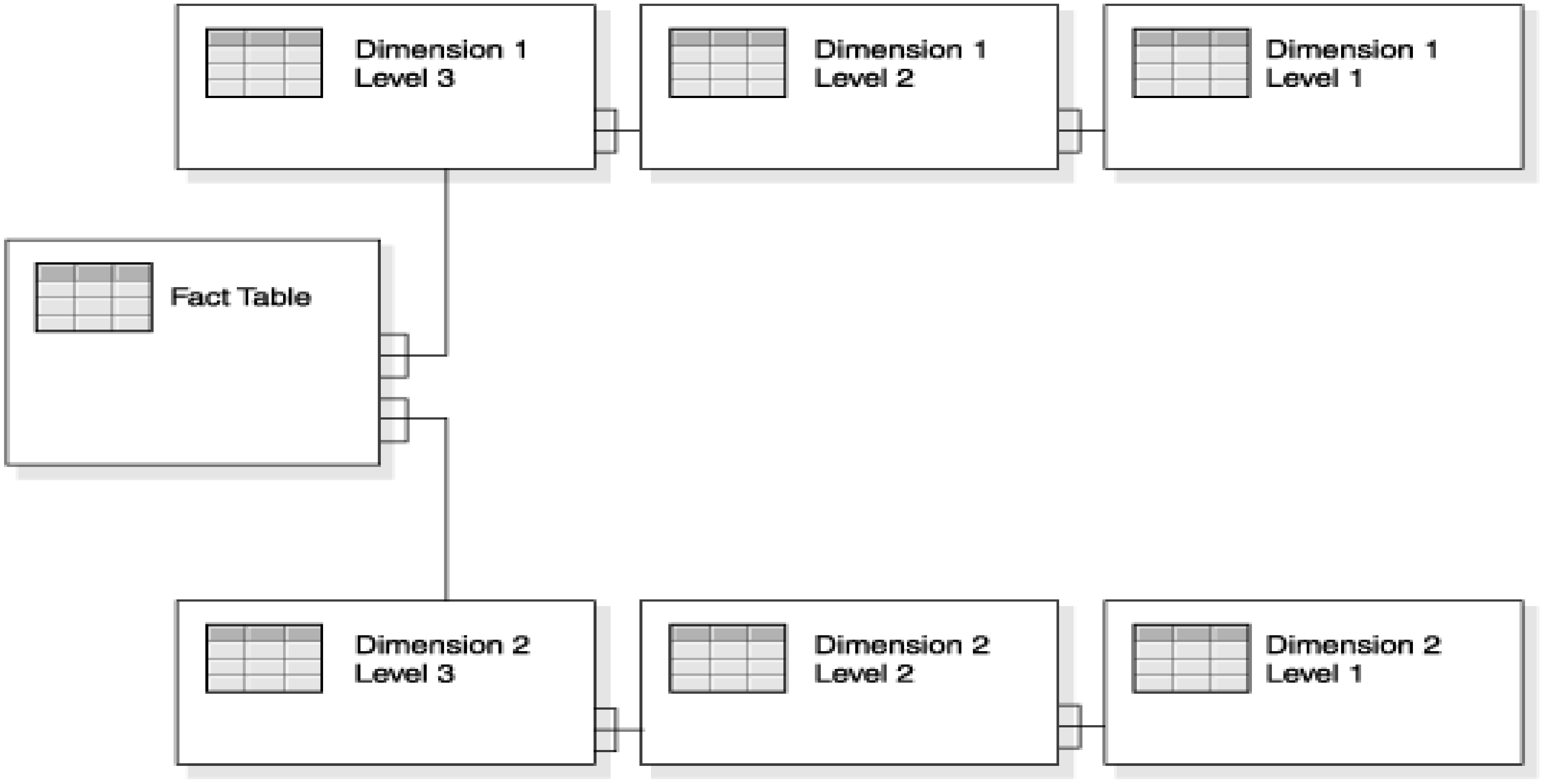
Hierarchical Structure: Data is stored in multiple levels to represent relationships (e.g., country → state → city).

Reduced Data Redundancy: Saves storage space but increases query complexity.

More Joins Required: Queries require more joins due to normalization, which can impact performance.

Used in Large Data Warehouses: Suitable for complex analytical queries where storage optimization is necessary.

Snowflake Schema



Example of a Snowflake Schema

Consider a **Sales Data Warehouse** with the following tables:

Fact Table: Sales_Fact (contains keys referencing dimensions and measures like sales amount, quantity).

Dimension Tables:

Date_Dim (date-related attributes)	Product_Dim (split into Product_Category and Product_Subcategory)	Customer_Dim (split into Customer, City, State, Country)

Comparison: Star Schema vs. Snowflake Schema

Feature	Star Schema	Snowflake Schema
Normalization	Denormalized	Normalized
Data Redundancy	High	Low
Query Performance	Faster (fewer joins)	Slower (more joins)
Storage Efficiency	Less efficient	More efficient
Complexity	Simple	Complex

When to Use Snowflake Schema?

- When storage optimization is a priority.
- When handling hierarchical data structures.
- When data consistency and integrity are important.
- When using automated ETL processes that can handle complex queries.

fact constellation schema

The **Fact Constellation Schema** (also known as a **Galaxy Schema**) is a complex data warehouse schema that consists of multiple fact tables sharing common dimension tables.

It is an extension of the **Star Schema** and **Snowflake Schema**, allowing for a more flexible and multi-faceted data representation.

Key Characteristics of Fact Constellation Schema

Multiple Fact Tables: Unlike the star and snowflake schemas, a fact constellation schema contains multiple fact tables for different business processes.

Shared Dimension Tables: Some dimension tables are shared among multiple fact tables, reducing redundancy.

Efficient for Complex Queries: Supports complex analytical queries across multiple business areas.

Higher Complexity: More complex structure with multiple joins, which may impact query performance.

Used in Large-scale Data Warehouses: Suitable for enterprises dealing with multiple business domains.

Example of a Fact Constellation Schema

Consider a **Retail Data Warehouse** with two fact tables:

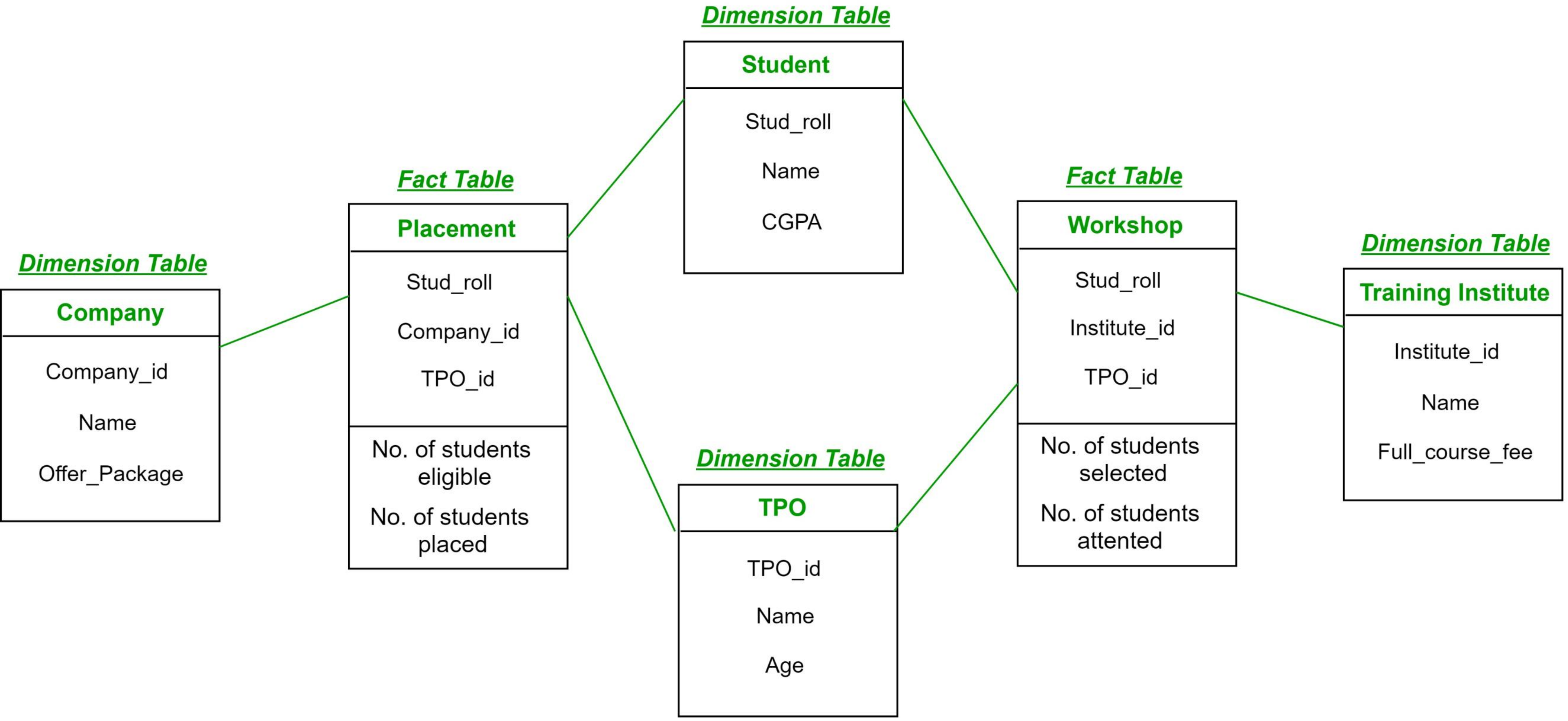
- **Sales_Fact** (for sales transactions)
- **Inventory_Fact** (for inventory tracking)

These fact tables share common dimensions:

- Date_Dim (date-related attributes)
- Product_Dim (product-related attributes)
- Store_Dim (store-related attributes)

However, they also have unique dimensions:

- Customer_Dim (specific to Sales)
- Supplier_Dim (specific to Inventory)


Dimension Table
Student
Stud_roll
Name
CGPA
Fact Table
Placement
Stud_roll
Company_id
TPO_id
No. of students eligible
No. of students placed
Dimension Table
Company
Company_id
Name
Offer_Package
Fact Table
Workshop
Stud_roll
Institute_id
TPO_id
No. of students selected
No. of students attented
Dimension Table
Training Institute
Institute_id
Name
Full_course_fee
Dimension Table
TPO
TPO_id
Name
Age

Comparison: Star Schema vs. Snowflake Schema vs. Fact Constellation Schema

Feature	Star Schema	Snowflake Schema	Fact Constellation Schema
Complexity	Simple	Moderate	High
Fact Tables	Single	Single	Multiple
Dimension Tables	Denormalized	Normalized	Shared across facts
Query Performance	Fast	Moderate	Slower (more joins)
Use Case	Small data marts	Medium-sized DW	Large-scale data warehouses

Handling Data Granularity

What is Data Granularity?

Data granularity indicates a logical table source's level of detail.

When a query is issued, the query engine uses the logical table source's data granularity to find the required level of detail for the requested data.

Data granularity measures how finely data is divided within a data structure.

Choosing the right level of data granularity is essential to ensure that your data analysis and predictions are accurate, your data is stored correctly, and that you can process it in the ways you want.

In effect, data granularity is critical to equipping you with the actionable insights needed to manoeuvre the decision-making process at work.

Types of data granularity

High (fine) granularity

- If you want your data to have high granularity, you will break down your data into very small levels of grain.
- An example of this would be recording keystrokes on a keyboard. In this case, each keystroke is a separate, distinct piece of data.
- This level of granularity is helpful for detailed analysis, such as audience segmentation or understanding user behaviour in a software application.

Intermediate granularity

- This type of granularity represents a middle ground, combining elements of both fine and coarse granularity.
- An example would be recording the times someone saved or edited their text.
- This level of granularity is more detailed than recording entire essays but less detailed than recording keystrokes.
- It’s useful for analyses where the complete detail is overwhelming, but some detailed insights are still needed.

Types of data granularity

Low (course) granularity

- Low granularity data is more summarized and consists of more extensive, aggregated units.
- If you recorded the final output, like an entire essay or submission, it would have coarse granularity.
- Here, you consider large blocks of data (each essay) without focusing on the finer details like sentences or words.
- This type of granularity is beneficial when the overall picture or summary is more important than the details.

Time-based granularity

- Time-based granularity can be coarse, fine, or intermediate, similar to how you can divide other data.
- The distinction is that time-based granularity refers to data categorized by specific time intervals.
- For example, if you collect data daily, in this scenario, you would combine all data for the day and then analyze it as a single unit.
- This type of granularity is ideal for trend analysis over time, such as weekly sales or performance metrics.

Why is data granularity important?

Enhanced Decision-Making:

Fine-grained data enables precise decision-making, especially in domains like finance, healthcare, and supply chain management.

Coarse-grained data helps in high-level strategic planning, such as market trend analysis.

Flexibility in Data Analysis:

Different analytical use cases require different levels of granularity.

Machine learning models and predictive analytics often need fine-grained data, whereas dashboards and reports typically work with aggregated data.

Improved Data Integration:

Matching data from multiple sources requires careful handling of granularity differences.

Ensuring consistent granularity enhances interoperability across datasets.

Why is data granularity important?

Optimized Query Performance:

Queries on highly detailed data can be slow and computationally expensive.

Pre-aggregating data or using indexing techniques improves efficiency in data retrieval.

Regulatory and Compliance Needs:

Certain industries, like finance and healthcare, require fine-grained data for audits and compliance.

Aggregating data can help anonymize sensitive information while still providing meaningful insights.

Effective Data Visualization:

Granularity affects how data is presented in charts and dashboards.

Coarse-grained data simplifies reporting, while fine-grained data allows for deeper drill-down capabilities.

Who uses data granularity?

Business decision-makers:	Leaders and managers use data at different granularity levels to make strategic decisions.
Public health professionals:	Data granularity significantly represents mortality or morbidity rates when assessing population health over time.
Financial professionals and accountants:	In areas like financial reporting, choosing the right level of granularity is vital for risk predictions and financial insights.
Health care professionals:	Time-based granularity is particularly important in health tracking, such as monitoring biological data over different time intervals.
Medical researchers:	Granularity is essential in medical research because it affects the detailed analysis of medical records and patient notes.

Working with Data Granularity

Data Granularity and Aggregate Tables in Oracle Analytics

1

Defining Data Granularity in Fact Tables

- Every fact table must have a specified granularity level.
- If no granularity is explicitly defined, Oracle Analytics assumes the most detailed level by default.
- Best practice: Always define granularity explicitly for each logical table source.

2

Defining Granularity for Dimension Tables

- Each logical table that joins a fact table must also have a defined granularity level.
- Ensures proper data aggregation and prevents inconsistencies during query execution.

About Aggregate Tables

Purpose of Aggregate Tables

- Store precomputed results from measures aggregated over specific dimensions.
- Improve query performance by reducing computational overhead.
- Example: A monthly sales table storing revenue sums for each product and store per month.

Joining Aggregate Fact and Dimension Tables

- Aggregated fact tables must be linked with appropriate dimension tables.
- Each aggregate table column represents a specific level of aggregation.

Logical Table Sources for Aggregate Fact Tables

- When creating an aggregate fact table source, corresponding dimension table sources must be created at the same aggregation level.
- At least one logical dimension source is required for each level of aggregation.
- If the dimension sources already exist, new ones are not required.

Example: Monthly Sales Fact Table

- **Fact Table: MonthlySalesFact**
  - Stores precomputed revenue sums per product, store, and month.
- **Required Dimension Table Sources:**
  1. **Product Dimension Source:**

     By logical level: ProductDimension.ProductLevel

     OR

     By column: Product.Product_Name

1. **Store Dimension Source:**

   By logical level: StoreDimension.StoreLevel

   OR

   By column: Store.Store_Name

2. **Time Dimension Source:**

   By logical level: TimeDimension.MonthLevel

   OR

   By column: Time.Month

Query Optimization in Oracle Analytics

The Oracle Analytics query engine selects the **most aggregated source** available that can answer a query.

This reduces query execution time and improves performance.

The most aggregated source is chosen based on the lowest multiplied number of elements (least detailed data).

About Aggregate Table Joins

You must create physical joins between the aggregate fact tables and the aggregate dimension tables.

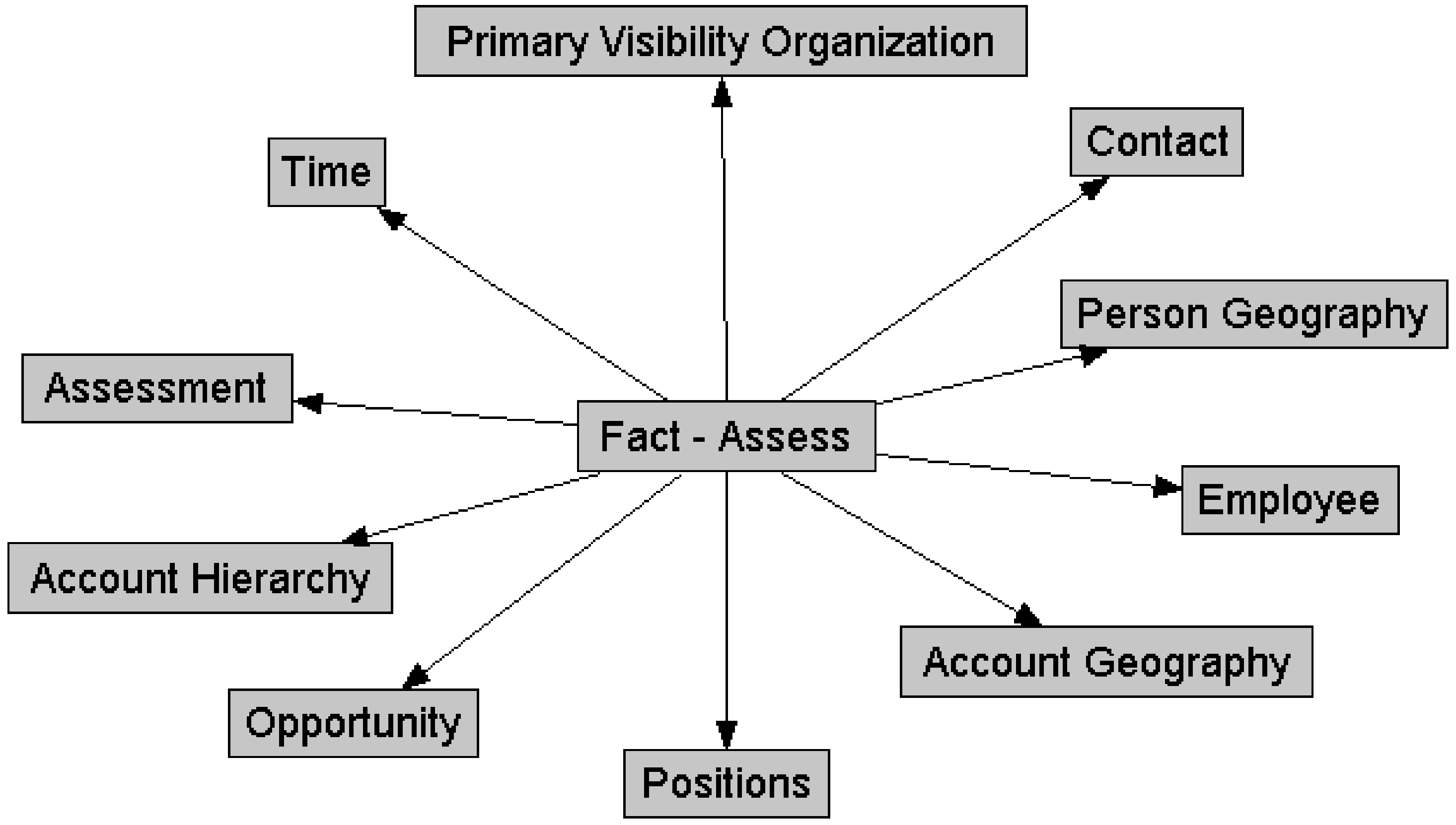
Joins tells the Oracle Analytics query engine where to send queries for physical aggregate fact tables joined to and constrained by values in the physical aggregate dimension tables.

You can verify joins by opening the fact logical table's logical diagram.

The diagram displays only the dimension logical tables that are directly joined to the fact logical table.

The diagram doesn't display dimension tables if the same physical table is used in logical fact and dimension sources

Aggregate Table Joins



Table

- The table contains a list of the logical level for each dimension table that's directly joined to the Fact - Assess fact table.

Dimension	Logical Level
Account Geography	Postal Code Detail
Person Geography	Postal Code Detail
Time	Day Detail
Account Organization	Account Detail
Opportunity	Opty Detail
Primary Visibility Organization	Detail
Employee	Detail
Assessment	Detail
Contact (W_PERSON_D)	Detail
FINS Time	Day
Positions	Details

Designing for query optimization

Best Practices for Query Optimization

- ✅ Use **Star Schema** for better performance.
- ✅ Define **Indexes** (Bitmap for low-cardinality, B-Tree for high-cardinality).
- ✅ **Partition large tables** (Range, List, Hash).
- ✅ Create **Materialized Views** for precomputed results.
- ✅ Use **SELECT only necessary columns** and filter data early with WHERE.
- ✅ Optimize **Joins** and avoid unnecessary joins.
- ✅ Pre-aggregate data to reduce query complexity.
- ✅ Use **Query Caching** and maintain **Database Statistics**.
- ✅ Enable **Parallel Query Processing** for large datasets.

Schema Design for Query Performance

Star Schema (Preferred for Query Performance)

- **Fact Table:** Contains numerical measures (e.g., sales revenue, order count).
- **Dimension Tables:** Contain descriptive attributes (e.g., customer, product, time).
- **Optimized for fast query performance due to simple joins.**
- Example:

```
FactSales (Sales_ID, Product_ID, Store_ID, Time_ID, Revenue, Quantity)
Product (Product_ID, Product_Name, Category, Brand)
Store (Store_ID, Store_Name, Location, Type)
Time (Time_ID, Date, Month, Year, Quarter)
```

- 🔹 **Why?** Fewer joins improve query speed.

Snowflake Schema (Normalized for Storage Efficiency)

- Breaks down dimensions into multiple related tables.
- Reduces redundancy but increases join complexity.
- Example:

```
Product → ProductCategory (Product_ID, Category_ID) → Category
(Category_ID, Category_Name)
```

- ◆ **Use case:** When storage is a bigger concern than query speed.

Indexing Strategies

Indexing Strategies

1. Bitmap Indexing (Best for Low-Cardinality Columns)

- Ideal for columns with few unique values (e.g., Gender, Product Category).
- Speeds up filtering in queries.
- Example:

```
CREATE BITMAP INDEX idx_gender ON Customer(Gender);
```

2. B-Tree Indexing (Best for High-Cardinality Columns)

- Works well for columns with many unique values (e.g., Order_ID, Customer_ID).
- Improves range-based searches.
- Example:

```
CREATE INDEX idx_order ON Orders(Order_Date);
```

Partitioning Strategies

Range Partitioning

- Divides tables based on a range of values (e.g., partitioning by year).
- Queries filtering on partitioned columns run faster.
- Example:

```
CREATE TABLE Sales (
        Sales_ID INT,
        Sale_Date DATE,
        Revenue DECIMAL(10,2) )
PARTITION BY RANGE (Sale_Date) (
        PARTITION sales_2023 VALUES LESS THAN (TO_DATE('2024-01-01', 'YYYY-MM-DD')),
        PARTITION sales_2024 VALUES LESS THAN (TO_DATE('2025-01-01', 'YYYY-MM-DD’))
);
```

List Partitioning

- Divides data based on predefined values (e.g., region-wise partitioning).
- Example:

```
PARTITION BY LIST (Region) (
  PARTITION north VALUES ('North America’),
  PARTITION south VALUES ('South America’)
  );
```

Hash Partitioning

•Distributes rows evenly using a hashing function to avoid data skew.
•Useful for evenly distributing large fact tables.
Example:

```
PARTITION BY HASH (Customer_ID) PARTITIONS 4;
```

Materialized Views for Query Caching

- Precompute and store query results to speed up performance.
- Ideal for aggregations, joins, and complex calculations.
- Example:

```
CREATE MATERIALIZED VIEW mv_monthly_sales AS
SELECT Product_ID, Store_ID, Time_ID, SUM(Revenue) AS Total_Revenue
FROM FactSales
GROUP BY Product_ID, Store_ID, Time_ID;
```

- 🔷 **Benefit:** Queries on FactSales can now use the precomputed results.

Query Optimization Techniques

Use SELECT Only for Needed Columns

- Avoid SELECT * as it fetches unnecessary data, slowing down queries.
- ❌ **Bad Query:**

```
SELECT * FROM FactSales;
```

- ✅ **Optimized Query:**

```
SELECT Product_ID, Store_ID, SUM(Revenue) FROM FactSales WHERE
Time_ID = '2024-01' GROUP BY Product_ID, Store_ID;
```

Use WHERE Clause for Filtering Early

- Filtering early in the query execution reduces data scanned.
- ❌ **Bad Query:**

```
SELECT Product_ID, Store_ID, SUM(Revenue) FROM FactSales GROUP BY
Product_ID, Store_ID HAVING Time_ID = '2024-01';
```

- ✅ **Optimized Query:**

```
SELECT Product_ID, Store_ID, SUM(Revenue) FROM FactSales WHERE
Time_ID = '2024-01' GROUP BY Product_ID, Store_ID;
```

- 🔹 **Why?** The WHERE clause reduces data before aggregation.

Use JOINs Efficiently

- Prefer **inner joins** over outer joins when possible.
- Avoid unnecessary joins by selecting only required columns.
- Example:

```
SELECT f.Product_ID, p.Product_Name, SUM(f.Revenue)
FROM FactSales f
JOIN Product p ON f.Product_ID = p.Product_ID
WHERE f.Time_ID = '2024-01’
GROUP BY f.Product_ID, p.Product_Name;
```

Data Aggregation and Summarization

- **Pre-aggregated Tables:** Store summarized data at different levels (e.g., daily, monthly, yearly).
- **ETL Processing:** Aggregate data before loading into the data warehouse.
- Example:

```
CREATE TABLE MonthlySales AS
SELECT Product_ID, Store_ID, Time_ID, SUM(Revenue) AS Total_Revenue
FROM FactSales
GROUP BY Product_ID, Store_ID, Time_ID;
```

- 🔹 **Benefit:** Queries on monthly sales will be much faster than scanning the entire FactSales table.

Caching and Performance Tuning

- Enable **Query Caching** to store frequently accessed query results.
- Use **Database Statistics** (e.g., ANALYZE TABLE) to help the optimizer choose the best execution plan.
- Example:

```
EXEC DBMS_STATS.GATHER_TABLE_STATS('FactSales');
```

- 🔹 **Why?** Keeps execution plans up to date for better query performance.

Load Balancing and Parallel Processing

- **Sharding:** Distribute large datasets across multiple servers for parallel query execution.
- **Parallel Query Execution:** Utilize multi-threading to process queries faster.
- Example:

```
ALTER SESSION ENABLE PARALLEL DML;

SELECT /*+ PARALLEL(4) */ SUM(Revenue) FROM FactSales WHERE Time_ID =
'2024-01';
```

Design Decisions

Choosing the Process	Selecting the subjects from the information packages for the first set of logical structures to be designed.
Choosing the Grain.	Determining the level of detail for the data in the data structures.
Identifying and Conforming the Dimensions.	Choosing the business dimensions (such as product, market, time, etc.) to be included in the first set of structures and making sure that each particular data element in every business dimension is conformed to one another.
Choosing the Facts.	Selecting the metrics or units of measurements (such as product sale units, dollar sales, dollar revenue, etc.) to be included in the first set of structures.
Choosing the Duration of the Database.	Determining how far back in time you should go for historical data.