Logical and Physical Data Modeling

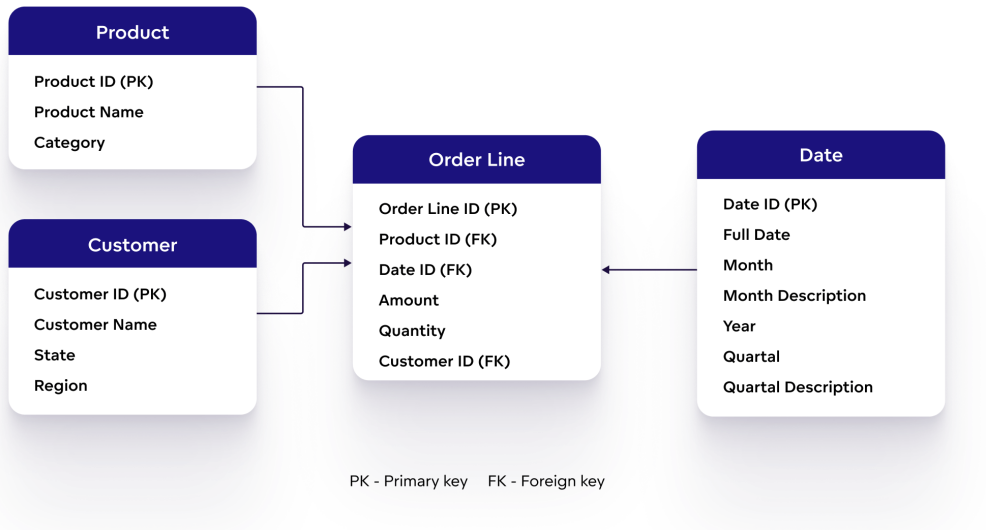
Definition of a Data Model:

A visual representation of data elements and their relationships, based on real-world objects.

Logical Data Model (LDM):

A **Logical Data Model** is an abstract representation of the data requirements of an organization, independent of any specific technology or database management system (DBMS).

- **Purpose**: Provides a detailed, structured description of data elements, entities, attributes, and their relationships.
- **Entities**: Objects relevant to business processes (e.g., Customers, Products).
- **Attributes**: Characteristics of entities (e.g., Customer Name, Product Price).
- **Use**: Created by business analysts and data architects to align business requirements with data structures.
- **Example**:
  - *E-commerce*: Data of customer orders is stored within the "Order Line" table, reusing previously stored customer and product data.
  - *Banking*: Customer accounts and their related purchases are stored in "Account" and "Purchase" entities.

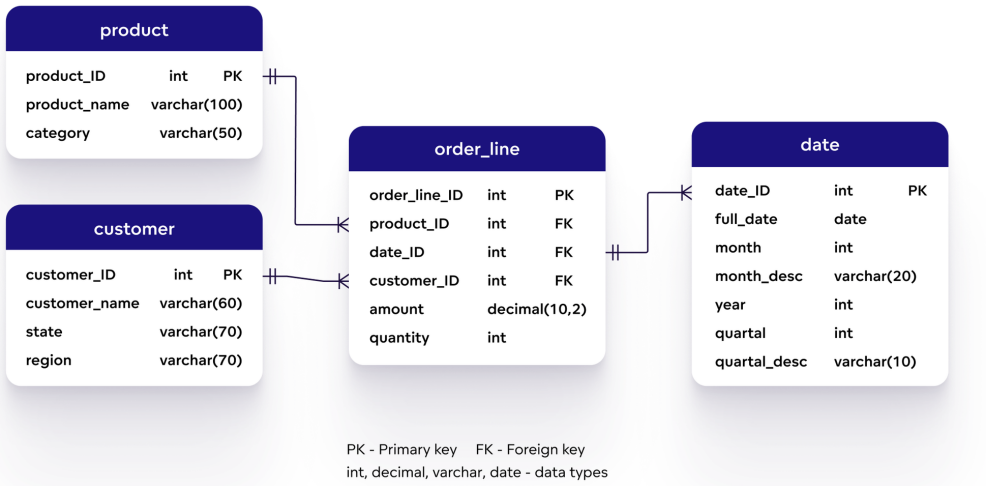


Physical Data Model (PDM):

A **Physical Data Model** translates the logical data model into a structure that can be implemented in a specific database management system (DBMS).

- **Purpose**: Specifies how data will be implemented in the database, translating LDM into technical details.
- **Components**: Table structures, column names, data types, constraints, primary/foreign keys, and indexes.
- **Responsibility**: Created by database administrators and developers to build the database.
- **Example**:

- *E-commerce*: Entities become tables, and attributes are translated into columns with specified data types.
- *Banking*: The LDM is transformed into physical tables storing customer and purchase information.



Transforming Normalized Data into Denormalized Data for Data Warehousing

Starting with Normalization:

In data warehousing, it's often beneficial to begin with a fully normalized design (usually at least to 3rd Normal Form) to ensure a clean, structured representation of the data. This provides a logical model where performance trade-offs and specific denormalization decisions can be evaluated effectively. From this clean starting point, denormalization decisions can be made during the physical design to improve performance.

There are a few reasons we would want to go through the process of normalization:

- Make the database more efficient
- Prevent the same data from being stored in more than one place (called an “insert anomaly”)
- Prevent updates being made to some data but not others (called an “update anomaly”)
- Prevent data not being deleted when it is supposed to be, or from data being lost when it is not supposed to be (called a “delete anomaly”)
- Ensure the data is accurate
- Reduce the storage space that a database takes up
- Ensure the queries on a database run as fast as possible

First Normal Form (1NF)

1st Normal Form ensures that the domains of attributes contain only atomic (simple, indivisible) values. Each value in a table must be atomic, meaning a single piece of data should not include multiple pieces of information or groupings.

1NF Rule:

- Each cell in the table must contain atomic value. This means that columns should not contain multiple values and composite attributes.
- No repeating groups or arrays are allowed in a table.
- Each column in a table must have a unique name.
- A column should contain value from the same domain.
- No Duplicate rows.
- No Duplicate Values in the Primary key.

Example (before normalization in 1NF):

Order ID	Product Information	Quantity
101	Laptop, Mouse	2, 1
102	Keyboard, Monitor, USB Cable	1, 2, 3

In this table, the **"Product Information"** and **"Quantity"** columns hold multiple values in a single column. This violates 1NF because they are not atomic values.

Example (after normalization in 1NF):

Order ID	Product	Quantity
101	Laptop	2
101	Mouse	1
102	Keyboard	1
102	Monitor	2
102	USB Cable	3

In the normalized form, each product and its quantity are represented in separate rows, ensuring atomic values for each attribute.

Indeed, another method to achieve the First Normal Form (1NF) is by splitting the original table into two separate tables.

Student_ID	Student_Name	Courses
1	John	Math, Physics
2	Alice	Chemistry, Biology, French
3	Bob	English

1. **Students Table**:

Student_ID	Student_Name
1	John
2	Alice
3	Bob

1. **Courses Table**:

Student_ID	Course
1	Math
1	Physics
2	Chemistry
2	Biology
2	French
3	English

Common Violations of 1NF:

1. **Multiple Values in a Single Attribute**:
   - **Violation**: When a column contains more than one value.
   - **Solution**: Split the values into separate rows or columns.

   Example (Violation):

Employee ID	Skills
001	Python, SQL
002	Java, HTML, CSS

   Solution (Normalized Table):

Employee ID	Skill
001	Python
001	SQL
002	Java
002	HTML
002	CSS

2. **Overloaded Attribute Values**:
   - **Violation**: An attribute contains values with multiple meanings.
   - **Solution**: Create separate attributes for each meaning and clarify the values.

   Example (Violation):

Account ID	Registration Info
123	Personal - Regular
456	Business - Premium

   Solution:
   Split into two separate columns, such as Account Type and Registration Level:

Account ID	Account Type	Registration Level
123	Personal	Regular
456	Business	Premium

3. **Repeating Group Structures**:
   - **Violation**: Having repeated groups in a table, such as several columns for months of sales.
   - **Solution**: Flatten the repeating groups into individual rows for flexibility in querying.

   Example (Violation):

Account #	Jan Balance	Feb Balance	Mar Balance
001	1000	1100	1200

   Solution:
   Convert into a normalized structure with a separate row for each month:

Account #	Month	Balance
001	Jan	1000
001	Feb	1100
001	Mar	1200

Benefits of 1NF in Data Warehousing:

- **Data Integrity**: Ensures that each piece of information is represented clearly without redundancy.
- **Flexibility**: By removing repeating groups, it becomes easier to query data.
- **Efficiency**: Atomic values enhance the performance of relational database systems by making indexing and searching faster.

Second Normal Form (2NF)

A table is in **Second Normal Form (2NF)** if it meets the requirements of the First Normal Form (1NF) and ensures that all non-prime attributes are fully functionally dependent on the **whole** primary key. A **non-prime attribute** is any attribute that is not part of the primary key. The primary goal of 2NF is to eliminate partial dependencies, where a non-prime attribute is dependent on only part of a composite key.

2NF Rule:

- The table must be in **1NF**.
- All non-prime attributes must be fully functionally dependent on the entire primary key (i.e., no partial dependencies on part of a composite key).

Example (Violation of 2NF):

Let's take an example where we have a table tracking the hours worked by employees on various projects, by date. The composite primary key is a combination of **SSN** (Social Security Number), **Project ID**, and **Date**.

SSN	Project ID	Date	Hours	Employee Name	Project Name
1234	101	2023-08-01	8	John Doe	AI Research
1234	101	2023-08-02	6	John Doe	AI Research
5678	102	2023-08-01	7	Jane Smith	Data Analysis

In this table:

- **SSN**, **Project ID**, and **Date** form the composite primary key.
- **Hours** depends on the entire composite key.
- **Employee Name** depends only on **SSN**, not on the whole composite key.
- **Project Name** depends only on **Project ID**, not on the whole composite key.

This violates 2NF because **Employee Name** and **Project Name** are only partially dependent on the composite primary key. This can lead to redundancy, as we repeatedly store the same employee and project information for each date.

Solution: Normalize to 2NF

To resolve the violation, we split the table into multiple tables, ensuring that all non-prime attributes are fully dependent on the entire primary key.

Table 1: Employee Table

Contains information about employees (SSN and Employee Name):

SSN	Employee Name
1234	John Doe
5678	Jane Smith

Table 2: Project Table

Contains information about projects (Project ID and Project Name):

Project ID	Project Name
101	AI Research
102	Data Analysis

Table 3: Employee_Project Table

Tracks the hours worked by employees on projects, with the composite primary key (SSN, Project ID, Date) and only attributes fully dependent on the whole key (Hours):

SSN	Project ID	Date	Hours
1234	101	2023-08-01	8
1234	101	2023-08-02	6
5678	102	2023-08-01	7

Key Insights and Fixes:

1. **Partial Dependency**:
   - **Violation**: Attributes depend only on part of a composite primary key.
   - **Fix**: Split the table into smaller, related tables to eliminate partial dependencies and redundancy.
2. **Referential Integrity**:
   - The employee and project tables act as reference tables. Every employee and project entry in the Employee_Project table must correspond to an existing entry in the Employee and Project tables, ensuring data integrity.
   - This structure allows for **better management** of employee and project information, reducing the likelihood of data anomalies.

Costs of 2NF (Additional Joins):

- The normalized design requires additional joins to retrieve related information (e.g., employee name, project name) from separate tables.
- When querying for employee hours, you will need to join the Employee_Project table with the Employee and Project tables.

Benefits of 2NF:

- **Reduced Redundancy**:

  Reduces the redundancy of employee and project information, leading to lower storage requirements.

- **Data Integrity**:

  Helps to avoid data anomalies, ensuring that the employee and project details are consistent across the database.

- **Scalability**:

  Having separate domain tables (for employees and projects) allows for the easy addition of new entries, even if no hours are being allocated to them. This improves the overall flexibility and scalability of the database.

Third Normal Form (3NF)

A table is in **Third Normal Form (3NF)** if it satisfies the conditions of **Second Normal Form (2NF)** and ensures that every non-prime attribute is **non-transitively dependent** on the primary key. In other words, no non-prime attribute should depend indirectly on the primary key via another non-prime attribute.

3NF Rule:

- The table must be in **2NF**.
- There should be no **transitive dependencies**: If a non-prime attribute depends on another non-prime attribute rather than directly on the primary key, it violates 3NF.

Example (Violation of 3NF):

Let's consider a shipment table where **Shipment#** is the primary key, and **Customer#** is a foreign key representing a customer. Additionally, the **Customer Name (Cust_Nm)** is stored in the table.

Shipment#	Customer#	Cust_Nm	Shipment_Date
S123	C001	John Doe	2023-08-01
S124	C002	Jane Smith	2023-08-02
S125	C001	John Doe	2023-08-03

In this table:

- **Shipment#** is the primary key.
- **Customer#** is fully dependent on **Shipment#**, so no problem there.
- **Cust_Nm** (Customer Name) is fully dependent on **Customer#**, not directly on **Shipment#**. This creates a **transitive dependency** because **Cust_Nm** is dependent on **Customer#**, which is dependent on **Shipment#**. As a result, this table violates the 3NF rule.

Solution: Normalize to 3NF

To resolve the transitive dependency, we should split the table into separate entities, ensuring that each non-prime attribute is directly dependent on the primary key.

Table 1: Shipment Table

Contains shipment-related information with **Shipment#** as the primary key:

Shipment#	Customer#	Shipment_Date
S123	C001	2023-08-01
S124	C002	2023-08-02
S125	C001	2023-08-03

Table 2: Customer Table

Contains customer information with **Customer#** as the primary key:

Customer#	Cust_Nm
C001	John Doe
C002	Jane Smith

By splitting the **Cust_Nm** into a separate **Customer Table**, we remove the transitive dependency. Now, each non-prime attribute depends only on the primary key of its respective table.

Key Insights and Fixes:

1. **Transitive Dependency**:
   - **Violation**: A non-prime attribute depends indirectly on the primary key (through another non-prime attribute).
   - **Fix**: Split the table into smaller, logically related tables to ensure that each attribute is directly dependent on the primary key.
2. **Elimination of Redundancy**:
   - By moving customer information into its own table, we reduce redundancy. For example, we no longer have to store "John Doe" multiple times for each shipment record.

Costs of 3NF:

- **Complexity in ETL Process**:
  - Extracting customer information into a separate entity requires sophisticated techniques such as fuzzy matching and may involve considerable data cleansing.
  - Consolidating multiple customer records into a single true customer record can be a complex task, requiring special tools like Trillium or Group One for accurate merging.
- **Additional Table Joins**:
  - Queries that need customer information alongside shipment data now require a join between the **Shipment Table** and the **Customer Table**. This adds processing overhead, but the tradeoff is often worth it due to reduced data redundancy.

Benefits of 3NF:

- **Reduced Storage Costs**:

  Storing customer details separately results in fewer redundant records.
- **Consistency**:

  By separating customer details, we ensure a unified and consistent view of each customer across the entire data warehouse.

- **Better Analysis**:

  Having a distinct **Customer Table** facilitates customer-centric analysis. For example, you can now answer questions like, “What percentage of our customers work in the consumer product goods industry?” without worrying about inconsistencies in the data.

Practical Consideration:

While pure normalization (like 3NF) is essential for data integrity and storage optimization, in real-world scenarios, some denormalization might be necessary for performance reasons. However, denormalization should be applied **selectively and cautiously** to avoid the risk of turning the database into a single, large, flat file.

Denormalization in Data Warehousing

Denormalization is the process of restructuring a database by introducing redundancy into a previously normalized schema to improve performance. In a normalized design, data is stored in multiple related tables, which requires frequent joins during queries. While normalization reduces redundancy and ensures data integrity, it may negatively impact query performance due to complex joins, especially in large databases like data warehouses.

Data warehouses prioritize **query performance** over strict data normalization. Since data warehouses are often queried for complex reports and analytics, denormalization is favored to speed up these queries by reducing the need for joins. In simple terms, denormalization trades some redundancy and increased storage for faster query execution.

Methods for Denormalization:

1. Pre-Join Denormalization

The **Pre-Join Denormalization** technique combines frequently joined tables into a single table to avoid the performance cost of joins during query execution. This approach is particularly useful when there's a **one-to-many** relationship between tables.

Example of Pre-Join Denormalization

Let’s use a **retail sales** example to explain this.

Before Denormalization (Normalized Structure):

- **Sales Header Table (Header)**

sale_id	store_id	sale_date
1001	S01	2023-08-30
1002	S02	2023-08-31

- **Sales Detail Table (Detail)**

tx_id	sale_id	product_id	quantity	price
1	1001	P01	2	50
2	1001	P02	1	30
3	1002	P03	3	45

In this normalized structure, the **Sales Header** table stores general information about each sale (store, date, etc.), and the **Sales Detail** table stores each individual item within that sale (products, quantity, price).

For queries that need data from both tables, such as “**What was the total sales volume on August 30, 2023, in Store S01?**”, the two tables must be joined, which can be computationally expensive if both tables have millions of rows.

After Denormalization (Pre-Join):

- **Sales Table (After Pre-Join)**

tx_id	sale_id	store_id	sale_date	product_id	quantity	price
1	1001	S01	2023-08-30	P01	2	50
2	1001	S01	2023-08-30	P02	1	30
3	1002	S02	2023-08-31	P03	3	45

In the denormalized table, the **store_id** and **sale_date** fields (from the Sales Header) are repeated in each detail row. This results in **redundancy**, but the benefit is that queries no longer need to join the Sales Header and Sales Detail tables.

Performance Impact:

1. **Improved Query Performance**:
   Queries like "
   What were the total sales for Store S01 on 2023-08-30?" can be answered directly from the denormalized table, without needing to join two large tables.
2. **Increased Storage Requirements**:
   Since
   store_id and **sale_date** are repeated for each product in the sale, the storage requirements increase. For example, if each sale had three products (a 1:3 ratio), the **store_id** and **sale_date** would be duplicated three times for each sale.

Joining Two Tables:

1. **If two tables are small:**

   There's no need for optimization because the cost of joining small tables is minimal.
2. **Join of two tables formula:**

   Join two tables Attributes = Table1 Attributes + Table2 Attributes − Common Attributes
3. **Size of the resulting table:**
   - The size depends on the number of rows and columns in both tables after the join.
   - Formula: Number of Rows * Number of Columns

Storage Cost Before and After Denormalization:

Assumptions:

Sales Header Table (Header) (Parent)

Use 30 byte and has 1 billion records. Assume sales_id is of 8 byte.

Sales Detail Table (Detail) (Child)

Use 40 byte and has 3 billion records.

Storage Cost Before Denormalization (Normalized):

Sales Header Table:

Storage = 1 billion records × 30 bytes = **30 GB**

Sales Detail Table:

Storage = 3 billion records × 40 bytes = **120 GB**

Total storage (normalized) = 30 GB + 120 GB = **150 GB**

Storage Cost After Denormalization:

Denormalized Record Size = 30 + 40 - 8 = **62 bytes**
Storage = 3 billion records × 62 bytes =
186 GB

Result:

- **Before Denormalization**: 150 GB.
- **After Denormalization**: 186 GB.
- **Increase in Size**: ((186−150) / 150)×100 ≈ 24%.

Performance Considerations:

There are different types of **joins** in databases, and each has a different impact on performance. When using pre-join denormalization, you avoid these joins and their performance costs:

- **Sort-Merge Join**: This join requires sorting data from both tables. By pre-joining, you avoid this overhead.
- **Hash Join**: This join splits large data into partitions. Pre-joining avoids the need for this operation.
- **Nested Loop Join**: This can result in repeated access to the same data, but pre-joining eliminates this inefficiency.

Considerations:

- **Storage Costs**: The size of the database increases due to repeated data. However, in data warehouses, the tradeoff of increased storage is acceptable if the query performance improves significantly.
- **Maintenance**: Denormalized tables are easier to query but harder to maintain. If a store location changes, it must be updated in every record, not just in the header table as it would be in a normalized design.
- **When to Use**: Pre-join denormalization is beneficial when querying large datasets where joins are common. However, it is less useful for small tables or cases where precise counts (e.g., distinct counts) are often needed.

Tradeoffs in Denormalization:

While denormalization improves query performance by reducing joins, it introduces several tradeoffs that need to be carefully considered:

1. **Performance Implications**:
   Denormalization typically speeds up read queries but may slow down writes due to redundant data updates.
2. **Storage Implications**:
   Since denormalization introduces redundancy, it increases the amount of data stored. In large data warehouses, this can be a significant cost.
3. **Ease-of-Use Implications**:
   Denormalized tables are easier to query, especially for non-technical users, as they do not need to understand complex joins.
4. **Maintenance Implications**:
   Maintaining consistency in denormalized data can be challenging, especially when updates are required across redundant fields.

2. Column Replication or Movement in Denormalization

Column replication or movement is a denormalization technique where frequently accessed columns from one table are either **replicated** (copied) or **moved** into another table. The goal is to avoid the overhead of performing joins between large tables during queries. This technique helps improve query performance but comes at the cost of increased storage requirements.

In some cases, rather than replicating columns into another table, you might move them entirely to avoid duplication. However, this should only be done when there is a **mandatory one-to-many relationship** between the two tables. This ensures that no important data is lost if a record in one table does not exist in the other.

Example: Health Care Data Warehouse

Let's look at a health care data warehouse example where we have two tables:

1. **Claim Header Table**: Contains general claim information.
   - claim_id
   - member_id
   - claim_date
2. **Claim Detail Table**: Contains individual items or services related to the claim.
   - detail_id
   - claim_id
   - service_description
   - cost

Normally, to get information such as a member's details for each claim, you would need to **join** the Claim Header and Claim Detail tables using `claim_id`.

Column Replication Example

To avoid the performance impact of repeatedly joining these tables, we can **replicate** the `member_id` from the **Claim Header** table into the **Claim Detail** table.

After Denormalization

detail_id	claim_id	service_description	cost	member_id
101	501	X-Ray	200	1001
102	501	Blood Test	50	1001

By replicating `member_id` into the Claim Detail table, we can avoid joining the tables each time we need information about the member associated with a claim. This saves **query time** but slightly increases storage because we're repeating the `member_id` for every detail in the Claim Detail table.

Performance Considerations

- **Join Savings**: Since columns like `member_id` are often used in queries, replicating them avoids joins, especially for large-scale joins between tables.
- **Storage Cost**: Every row in the **Claim Detail** table will now include an extra column (`member_id`), which increases storage requirements.
- **Performance Benefit**: The performance boost is more significant when avoiding joins for large tables. For small tables, the impact might be minimal.

This technique is similar to **pre-join denormalization**, but in this case, we're only replicating a specific column (or a few columns) rather than joining entire tables. **The trade-offs** remain similar—storage increases, but query performance improves, especially for frequently accessed columns.

Caveat: Mandatory One-to-Many Relationship

This method should only be used when there is a **mandatory** one-to-many relationship between the two tables. For example, every claim **must** have a corresponding member. However, if this relationship is not mandatory, it's better to **replicate** rather than move the column to ensure no data is lost.

3. Pre-aggregation

Pre-aggregation refers to the process of computing frequently used aggregate values in advance and storing them in physical tables. This method helps avoid performing the same aggregation on detailed data over and over again, improving performance for certain queries. For example, instead of calculating total sales or customer count each time a report is generated, the pre-calculated summaries are quickly retrieved from pre-aggregate tables.

Advantages:

- **Performance Improvement:** By pre-calculating commonly used values, queries can avoid the heavy computational cost of re-aggregating detailed data.
- **Storage Impact:** Typically, the space required for pre-aggregate tables is relatively small compared to the detailed data. However, if a large number of multi-dimensional summaries are built, storage needs can grow significantly.
- **Ease of Use:** Pre-aggregated data simplifies data access for users. Instead of running complex queries to compute totals or averages, users can directly query the pre-aggregated data.

Disadvantages:

- **Maintenance Overhead:** Maintaining pre-aggregated tables adds a significant burden. The more frequently data changes, the harder it becomes to keep the aggregate tables up to date. Each update requires recalculating the aggregates, which can be resource-intensive.

Typical Examples of Pre-aggregation in Different Industries:

- **Retail:** Pre-aggregated tables may store information like total sales revenue, inventory on hand, cost of goods sold, and quantity of goods sold by store, item, and week.
    - **Example:** Instead of recalculating weekly sales for every store in a retail chain, the total sales revenue for each store is pre-calculated and stored in a summary table. For a query asking for "Total sales for store #101 for Week 35," the system simply retrieves the pre-aggregated value from the summary table.
- **Healthcare:** Pre-aggregated values such as effective membership by age and gender, product type, network, and month.
    - **Example:** In healthcare, a query for "Total male members in the 40–50 age group for product A in January" can be answered by retrieving pre-aggregated membership data instead of scanning all member records and calculating the value each time.

Critical Considerations:

1. **Business Agreement on Definitions:** It's important to agree on standard definitions for aggregates (e.g., whether sales are based on calendar months or billing cycles) to avoid confusion and extra maintenance.
2. **Maintenance Overhead:** Pre-aggregated data needs regular updates when new data is added. This can be more expensive than updating detailed records, especially with frequent changes.

3. **Update Strategy:**
    - **Transactional Update:** Update only the rows that need changes.
    - **Re-build:** Recompute all aggregates when many records change.
    - **Example:** If a few sales records change, update them; if many records change, rebuild the whole table.
4. **Consistency:** Aggregates must match detailed data, so when detailed records are updated or deleted, aggregates must be recalculated.
5. **Frequency of Use:** Create pre-aggregated tables only if they are used often. Use views to test if a physical table is needed. If not frequently used, relying on views might be better.