Topics to be covered:

| Week | Broader Topic | Topics | Tools to be covered |
| --- | --- | --- | --- |
| 3 | Functional Dependencies and Normalization | 1. Introduction to Functional Dependencies 2. Normalization: Functional Dependencies 3. Decomposition of Full Functional Dependency (FFD) 4. Transitive Dependency 5. De-normalization |  |

1. Introduction to Functional Dependencies

In database management systems (DBMS), functional dependencies play a vital role in defining relationships between attributes within a relational database. Functional dependencies help to ensure data integrity and consistency by enforcing rules on how data should be stored and organized. They provide a formal framework for analyzing and understanding the dependencies between attributes.

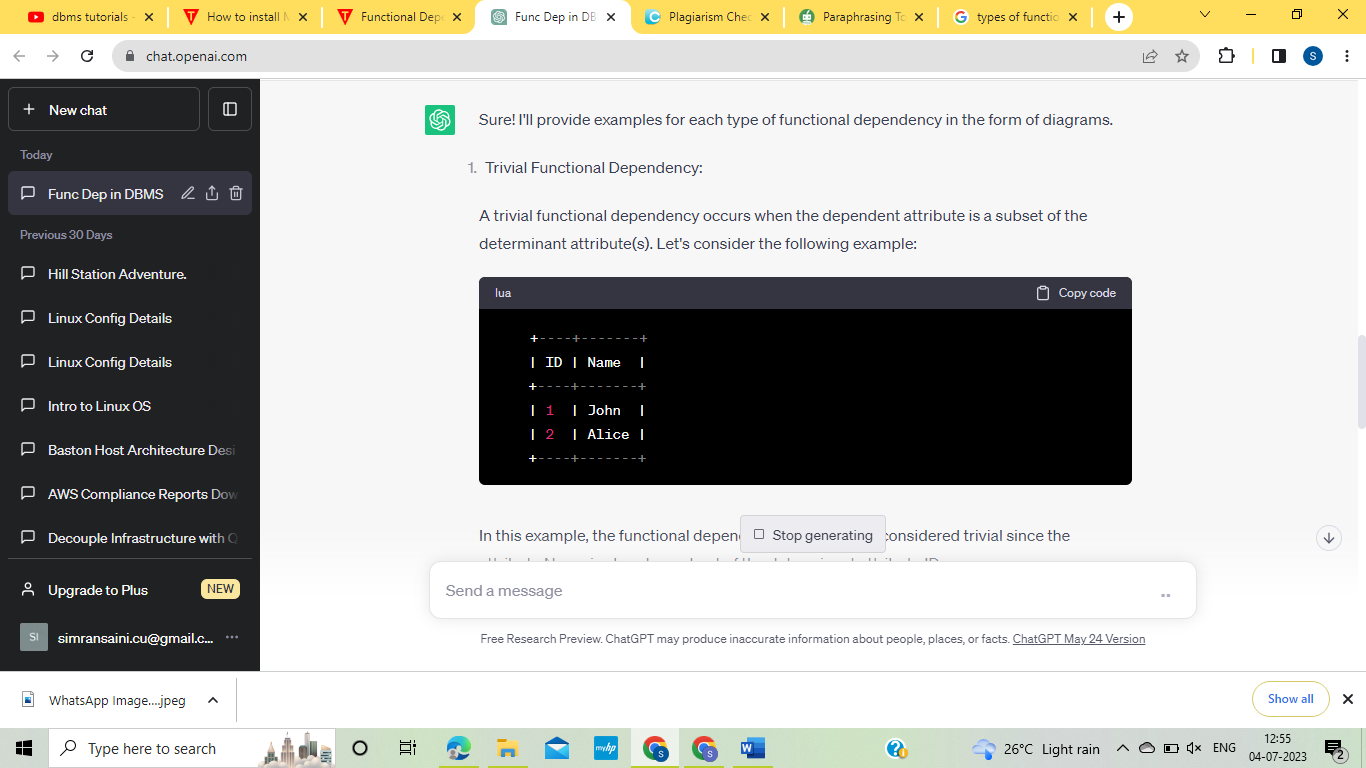
When the value of one or more attributes affects the value of another property, there is a functional dependency. It specifies the relationship between the attributes based on the concept of functional dependence. In simpler terms, if we know the values of certain attributes, we can determine the values of other attributes in a relational table.

A functional dependency is represented as X → Y, where X and Y are sets of attributes. It means that for every combination of values of attributes in X, there is a unique combination of values of attributes in Y. In other words, the value of Y is functionally dependent on the value of X.

Types of Functional dependencies:

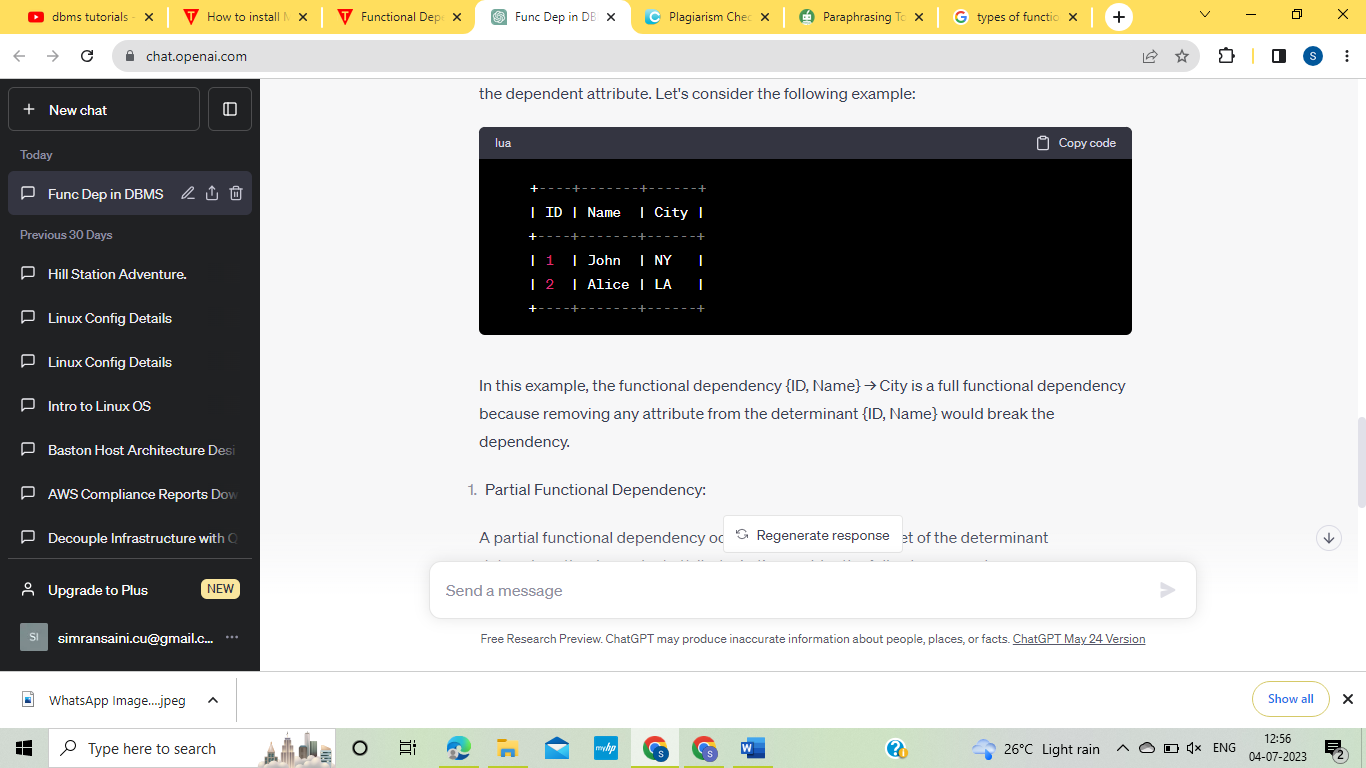
**There are various types of functional dependencies** that can exist in a relational database.

**1. Trivial Functional Dependency**: A functional dependency **X → Y** is considered trivial if Y is a subset of X. In this case, the dependency is implicit and does not provide any new information.



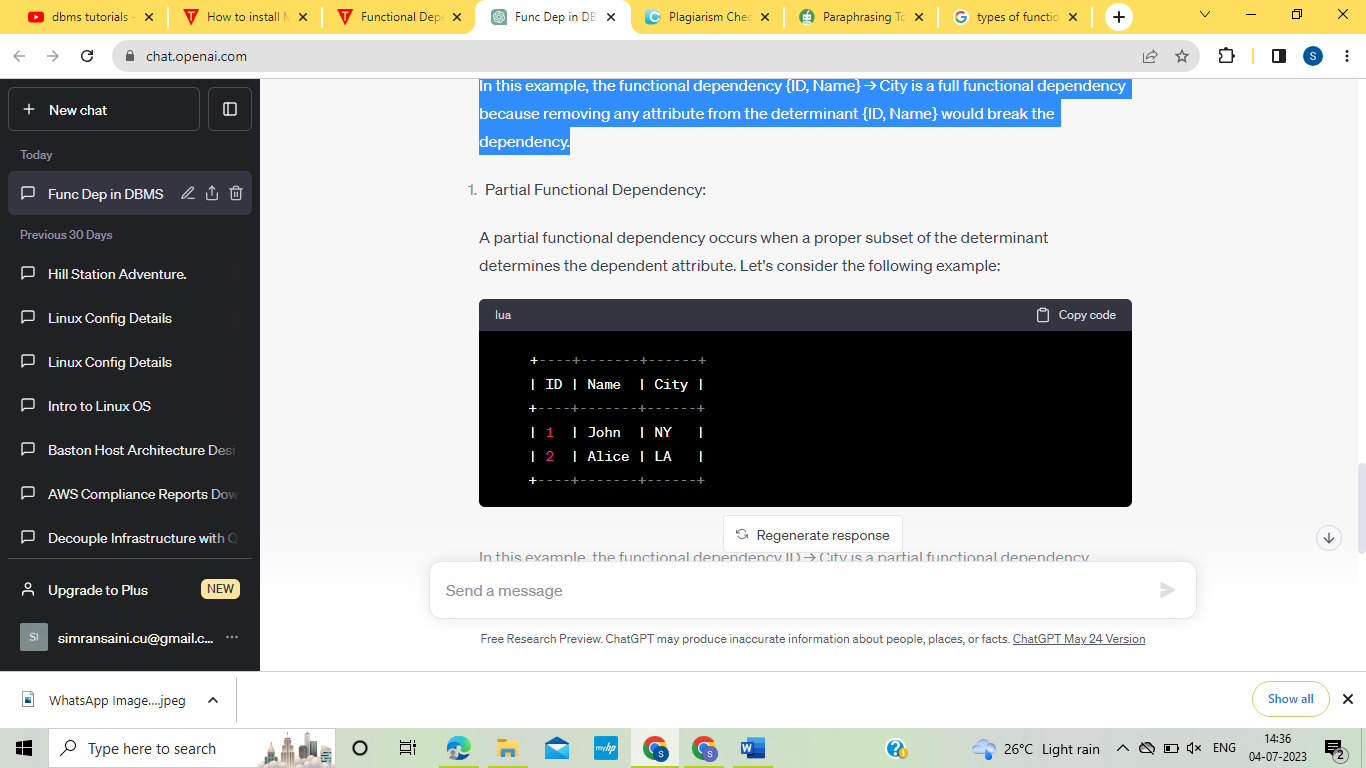
In this example, the functional dependency ID → Name is considered trivial since the attribute Name is already a subset of the determinant attribute ID.

**2. Full Functional Dependency:** A functional dependency **X → Y** is considered a full functional dependency if removing any attribute from X would break the dependency. It implies that no subset of **X** can determine **Y**.



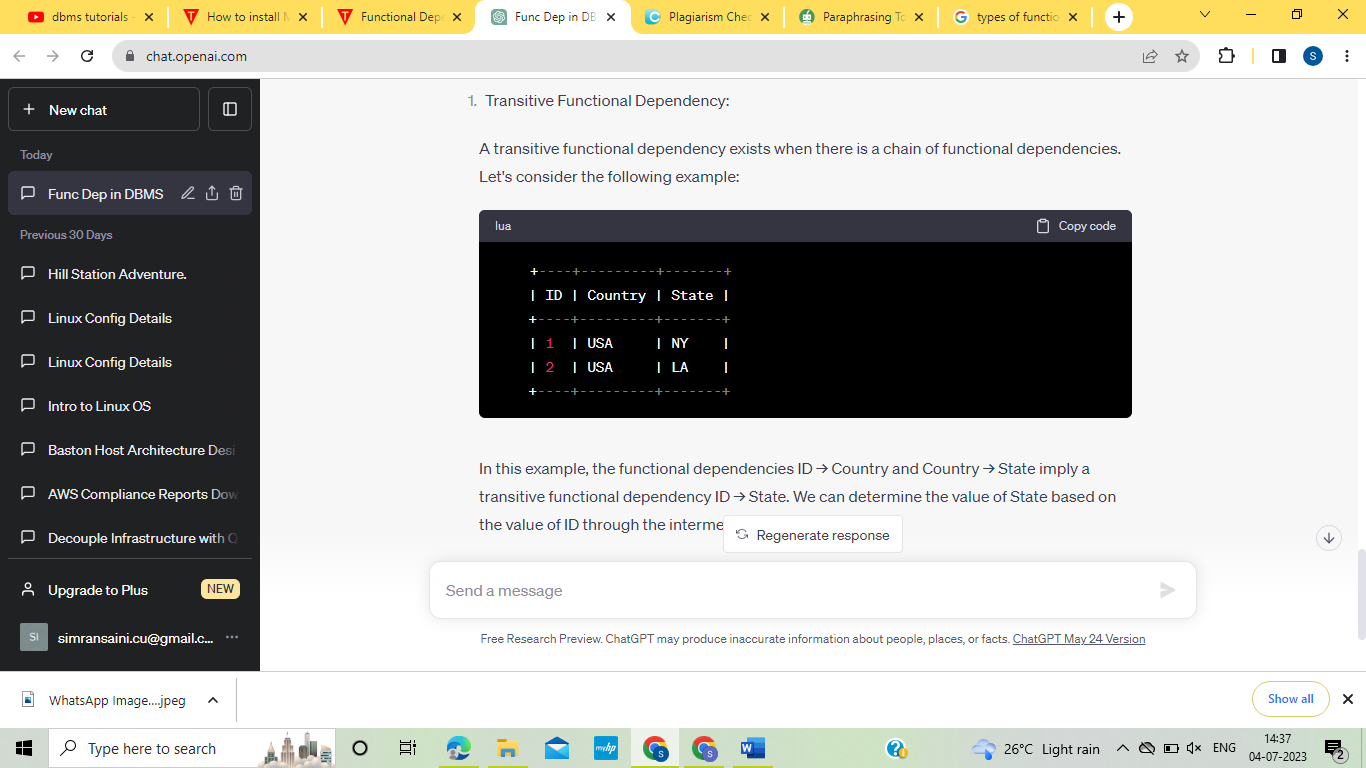
In this example, the functional dependency {ID, Name} → City is a full functional dependency because removing any attribute from the determinant {ID, Name} would break the dependency.

**3. Partial Functional Dependency**: A functional dependency **X → Y** is considered a partial functional dependency if removing one or more attributes from **X** would not break the dependency. In other words, a proper subset of **X** can determine **Y**.



In this example, the functional dependency ID → City is a partial functional dependency because removing the attribute Name from the determinant {ID, Name} would not break the dependency.

**4. Transitive Functional Dependency:** A functional dependency **X → Y** and **Y → Z** imply a transitive functional dependency **X → Z**. It means that if the values of attributes in **X** determine the values of attributes in **Y**, and the values of attributes in **Y** determine the values of attributes in **Z**, then the values of attributes in **X** determine the values of attributes in **Z**.



In this example, the functional dependencies ID → Country and Country → State implies a transitive functional dependency ID → State. We can determine the value of State based on the value of ID through the intermediate attribute Country.

Functional dependencies are essential for database design and normalization. By analyzing the functional dependencies in a relation, we can identify potential anomalies and redundancies in the database schema. The process of decomposing a relation into smaller, well-structured relations while preserving the functional dependencies is known as normalization.

Normalization helps eliminate data redundancy and improves data integrity. It also aids in efficient query processing and maintenance of the database system. By understanding and utilizing functional dependencies, database designers can create more robust and efficient database schemas.

2. Normalization: Functional Dependencies

Normalization is a process used in database design to organize data efficiently and eliminate redundancy, anomalies, and inconsistencies. It involves decomposing a relation (table) into smaller, well-structured relations while preserving the functional dependencies between attributes. Functional dependencies play a crucial role in normalization, as they define the relationships between attributes and guide the decomposition process.

Functional dependencies describe the relationships between attributes in a relation. A functional dependency **X → Y** means that for every combination of values of attributes in **X**, there is a unique combination of values of attributes in **Y**. In simpler terms, the value of **Y** is functionally dependent on the value of **X**.

Let's explore the various normal forms based on functional dependencies:

**1. First Normal Form (1NF):**

- Ensures atomicity: Each attribute in a relation contains only indivisible (atomic) values.

- No repeating groups: There should be no duplicate rows or multiple values in a single attribute.

- No ordering: The order of tuples and attributes doesn't matter.

Consider the following relation representing student grades:

Student (Student ID, Name, Course, Grades)

To convert it into 1NF, we ensure atomicity and eliminate repeating groups:

Student (Student ID, Name, Course, Grade)

**2. Second Normal Form (2NF):**

- Satisfies 1NF.

- No partial dependencies: Every non-key attribute is fully functionally dependent on the primary key. In other words, no proper subset of the primary key determines a non-key attribute.

- If an attribute depends on only part of the primary key, it should be moved to a separate relation.

Assuming the following functional dependencies exist:

Student ID, Course → Grade

Course → Instructor

The relation Student can be decomposed into two relations:

Student (Student ID, Name)

Course (Course, Instructor)

**3. Third Normal Form (3NF):**

- Satisfies 2NF.

- No transitive dependencies: No non-key attribute should be functionally dependent on another non-key attribute.

- If a non-key attribute depends on another non-key attribute, it should be moved to a separate relation.

Assuming the following functional dependencies exist:

Student ID → Name, Address

Course → Instructor

The relation Student can be further decomposed:

Student (Student ID, Name)

Address (Student ID, Address)

Course (Course, Instructor)

**4. Boyce-Codd Normal Form (BCNF):**

- Satisfies 3NF.

- No non-trivial dependencies: Every determinant (left side of the functional dependency) is a superkey (determines all attributes).

- If a non-key attribute determines another non-key attribute, it should be moved to a separate relation.

Assuming the following functional dependencies exist:

Student ID → Name

Course → Instructor

The relation Student can be decomposed:

Student (Student ID, Name)

Course (Course, Instructor)

**5. Fourth Normal Form (4NF):**

- Satisfies BCNF.

- No multi-valued dependencies: No non-key attribute should determine a set of attributes that have independent relationships with the primary key.

- If there are multi-valued dependencies, they should be moved to separate relations.

Assuming the following functional dependencies exist:

Student ID → Name, Address

Course → Instructor

Course → Textbook

The relation Student can be decomposed:

Student (Student ID, Name)

Address (Student ID, Address)

Course (Course, Instructor)

Textbook (Course, Textbook)

These are the most commonly used normal forms based on functional dependencies. It's important to note that achieving higher normal forms requires more rigorous decomposition and can result in a higher number of relations. The choice of the appropriate normal form depends on the specific requirements of the database and the trade-offs between simplicity, efficiency, and data integrity.

By applying normalization techniques based on functional dependencies, database designers can create well-structured, efficient, and maintainable database schemas that accurately represent the underlying data model while minimizing redundancy and anomalies.

3. Decomposition of Full Functional Dependency (FFD):

In the context of normalization in database design, decomposition refers to the process of breaking down a relation (table) into multiple smaller relations to eliminate redundancy and achieve a higher normal form. When decomposing a relation, one of the key considerations is preserving the full functional dependencies (FFDs) that exist within the original relation.

A full functional dependency exists when no proper subset of the determinant determines the dependent attribute. In other words, removing any attribute from the determinant would break the dependency. FFDs play a crucial role in database normalization because they help ensure that each attribute in a relation is functionally dependent on the entire primary key.

To decompose a relation while preserving FFDs, we follow these steps:

1. Identify the FFDs: Analyze the functional dependencies in the original relation to identify the full functional dependencies. A functional dependency X → Y is a full functional dependency if removing any attribute from X would break the dependency.

2. Create separate relations for FFDs: For each FFD identified, create a new relation with the attributes from the determinant and the dependent attribute(s). Include the primary key attributes from the original relation in each new relation to maintain the referential integrity.

3. Copy necessary attributes: If other attributes are functionally dependent on the primary key but not part of any FFD, copy them into the new relations as well.

4. Update references: Update the references to the new relations in any related relations, maintaining referential integrity.

**Example of decomposition of a relation :**

Consider the following relation representing students and their courses:

Student\_Course (Student ID, Student Name, Course ID, Course Name, Instructor)

Assuming the functional dependency {Student ID, Course ID} → {Student Name, Course Name}, we have a full functional dependency. To decompose this relation while preserving the FFD, we create two new relations:

Student (Student ID, Student Name)

Course (Course ID, Course Name, Instructor)

The resulting decomposed relations maintain the FFD, as the determinant {Student ID, Course ID} is preserved in the **Student\_Course** relation, and the dependent attributes {Student Name, Course Name} are moved to the respective new relations.

The decomposition process ensures that each attribute in the new relations is functionally dependent on the entire primary key, preventing redundancy and anomalies. By decomposing the relation based on FFDs, we achieve a higher normal form and improve data integrity and consistency in the database schema.

It's important to note that decomposition based on FFDs is just one aspect of the normalization process, and other functional dependencies and normal forms must also be considered for a comprehensive normalization of the database schema.

**There are two main types of decomposition:** *Lossless decomposition* and *Dependency-preserving decomposition*. Let's explore each type in detail:

**1. Lossless Decomposition:**

Lossless decomposition refers to the process of breaking down a relation into multiple smaller relations while ensuring that the original information can be reconstructed without any loss. In other words, the decomposition retains all the functional dependencies present in the original relation, and no information is lost during the decomposition process.

To achieve lossless decomposition, we aim to decompose the relation in such a way that the resulting smaller relations can be joined back together using a common attribute or set of attributes. This ensures that the original data can be reconstructed from the decomposed relations.

Lossless decomposition is important in database design to maintain data integrity and consistency. It ensures that the decomposition process does not introduce any inconsistencies or contradictions in the data.

Consider the following relation:

Employee (Employee ID, Name, Department)

To achieve lossless decomposition, we can decompose the relation into two smaller relations based on a common attribute:

Employee\_1 (Employee ID, Name)

Employee\_2 (Employee ID, Department)

By decomposing the relation in this way, we can join the two smaller relations using the common attribute "Employee ID" to reconstruct the original relation without any loss of information.

**2. Dependency-Preserving Decomposition:**

Dependency-preserving decomposition refers to the process of breaking down a relation into multiple smaller relations while preserving the functional dependencies that exist in the original relation. In other words, the decomposed relations should still exhibit the same functional dependencies as the original relation.

During dependency-preserving decomposition, we analyze the functional dependencies in the original relation and ensure that each decomposed relation retains the necessary attributes to maintain those dependencies. This ensures that the integrity and meaning of the data are preserved after the decomposition.

Preserving functional dependencies is crucial in database design as it ensures that the database maintains data consistency and accuracy. It helps prevent update anomalies and ensures that the decomposed relations can be used effectively in querying and updating the database.

Consider the following relation:

Order (Order ID, Customer ID, Product ID, Quantity)

Assuming the functional dependencies are:

Order ID → Customer ID

Order ID, Product ID → Quantity

To achieve dependency-preserving decomposition, we can decompose the relation into three smaller relations while preserving the functional dependencies:

Order\_1 (Order ID, Customer ID)

Order\_2 (Order ID, Product ID)

Order\_3 (Order ID, Quantity)

By decomposing the relation in this way, each smaller relation retains the necessary attributes to maintain the functional dependencies present in the original relation.

Both lossless decomposition and dependency-preserving decomposition are desirable in database design. While it may not always be possible to achieve both simultaneously, database designers aim to strike a balance between the two to ensure a well-structured and efficient database schema.

By considering lossless decomposition and dependency preservation during the decomposition process, database designers can create a normalized database schema that is free from redundancy, anomalies, and inconsistencies while preserving the necessary relationships between attributes.

4. Transitive Dependency

Transitive Dependency in Database Management Systems:

In the context of database management systems and normalization, a transitive dependency occurs when an attribute is functionally dependent on another attribute through a chain of functional dependencies. In other words, there is an intermediate attribute that mediates the dependency between two other attributes. Transitive dependencies can lead to data redundancy and update anomalies if not properly handled.

Let's explain transitive dependency with an example using a table with at least five entries:

Consider the following table representing employees and their projects:

Employee\_Project (Employee ID, Employee Name, Project Name, Project Manager)

Assume the following data:

| Employee ID | Employee Name | Project Name | Project Manager |

|-------------|---------------|--------------|-----------------|

| 101 | Alice | Project A | John |

| 102 | Bob | Project B | Alice |

| 103 | John | Project C | Bob |

| 104 | Mary | Project A | John |

| 105 | Mark | Project D | Bob |

In this table, Employee ID is the primary key, and Employee Name, Project Name, and Project Manager are attributes.

Transitive Dependency Example:

We can observe a transitive dependency between the attributes Employee Name and Project Manager. The functional dependencies are:

1. Employee ID → Employee Name (Direct Dependency)

2. Project Name → Project Manager (Direct Dependency)

However, we can deduce another functional dependency through the chain of Employee ID → Employee Name → Project Manager:

3. Employee ID → Project Manager (Transitive Dependency)

Here's how the transitive dependency works:

1. For Employee ID 101 (Alice), we can find the Project Manager by first finding Employee Name (Alice) and then looking up the Project Manager for Project Name (Project A), which is John.

2. For Employee ID 102 (Bob), we find the Project Manager through the intermediate attribute Employee Name (Bob), and then we find the Project Manager for Project Name (Project B), which is Alice.

Transitive dependencies can lead to data redundancy and update anomalies. For example, if the Project Manager for Project A changes from John to Mary, we need to update all occurrences of John to Mary in the table, which could be error-prone and time-consuming.

Normalization to Handle Transitive Dependency:

To handle transitive dependency, we can perform normalization and decompose the original table into smaller, well-structured relations. By doing so, we eliminate the need to store redundant information and prevent update anomalies.

We can decompose the Employee\_Project table into the following two tables:

Employee (Employee ID, Employee Name)

Project (Project Name, Project Manager)

Now, the transitive dependency is resolved as follows:

1. Employee ID → Employee Name (in the Employee table)

2. Project Name → Project Manager (in the Project table)

With this decomposition, we can avoid storing the Employee Name and Project Manager redundantly in multiple rows, which leads to a more efficient and normalized database design.

Transitive dependency occurs when an attribute is functionally dependent on another attribute through an intermediate attribute. This can lead to data redundancy and update anomalies. By properly identifying and resolving transitive dependencies through normalization, we can create a well-structured and efficient database schema.

5.De-normalization

De-normalization is the process of intentionally introducing redundancy into a database design to improve query performance or simplify data retrieval. It involves relaxing the normalization principles and deliberately duplicating data in order to optimize certain aspects of database performance, such as faster data retrieval, reduced join operations, or improved reporting capabilities. De-normalization is typically employed in situations where the benefits of denormalization outweigh the potential drawbacks of increased redundancy and complexity.

Here are some key points and considerations regarding de-normalization:

1. Performance Optimization: De-normalization is primarily used to improve the performance of database queries. By storing redundant data, the need for complex joins and costly calculations can be minimized, resulting in faster query execution times.

2. Read-Heavy Applications: De-normalization is often beneficial in read-heavy applications where the majority of operations involve data retrieval rather than updates. By pre-computing and storing data redundantly, the application can reduce the need for complex and resource-intensive query operations.

3. Simplified Data Model: De-normalization can simplify the data model by eliminating the need for excessive joins and complex relationships between tables. This can lead to easier data retrieval and simpler queries, making it more intuitive for developers and analysts.

4. Aggregated Data: De-normalization is commonly used when working with aggregated data, such as summary reports or analytics. By pre-calculating and storing aggregated values, the system can quickly retrieve the desired results without needing to perform extensive calculations on the fly.

5. Denormalization Techniques: There are several denormalization techniques commonly used, such as duplicating data across tables, introducing derived attributes, creating summary tables, and using materialized views. Each technique is applied based on the specific requirements and performance goals of the application.

6. Trade-Offs: De-normalization introduces redundancy and increases data storage requirements. This can lead to data inconsistency and update anomalies if not managed properly. It requires careful consideration and monitoring to ensure that data remains consistent across redundant copies.

7. Maintenance and Data Integrity: When employing de-normalization, extra attention must be given to maintaining data integrity. Any updates, inserts, or deletes need to be carefully handled to ensure that redundant data remains consistent. This may involve additional triggers, procedures, or application logic to manage the synchronization of data.

8. Scalability and Flexibility: De-normalization can impact the scalability and flexibility of the database design. Changes to the structure or relationships between data may require updates across multiple redundant copies, making the maintenance and evolution of the system more complex.

9. Use with Caution: De-normalization should be approached with caution, and the decision to denormalize should be based on a thorough analysis of the application's specific requirements, performance bottlenecks, and trade-offs. It is not recommended as a general practice for all database designs.

De-normalization is a technique employed in certain scenarios to optimize performance and simplify data retrieval in read-heavy applications. It involves deliberately introducing redundancy into the database design, which requires careful management to maintain data integrity. Understanding the specific needs and trade-offs of the application is crucial in deciding when and how to appropriately use de-normalization.

**An example demonstrating de-normalization in tabular form:**

Consider a normalized database schema for an e-commerce application with two tables: `Customers` and `Orders`. The normalized schema is as follows:

Customers (Customer ID, Customer Name, Address, City, State, Zip)

Orders (Order ID, Customer ID, Order Date, Total Amount)

Now, let's introduce de-normalization to improve query performance for certain scenarios. We can denormalize the schema by duplicating the `Customer Name` attribute from the `Customers` table into the `Orders` table:

Customers (Customer ID, Customer Name, Address, City, State, Zip)

Orders (Order ID, Customer ID, Customer Name, Order Date, Total Amount)

By duplicating the `Customer Name` attribute in the `Orders` table, we avoid the need to perform a join operation between `Customers` and `Orders` to retrieve the customer name when querying order information. This can significantly improve the performance of queries that involve retrieving order details along with the customer name.

Here's a sample dataset illustrating the denormalized schema:

```

**Customers Table:**

| Customer ID | Customer Name | Address | City | State | Zip |

|------------------|-----------------------|------------------|----------|---------|----------------|

| 101 | John Smith | 123 Main St. | New York | NY | 10001 |

| 102 | Alice Johnson | 456 Elm St. | Los Angeles | CA | 90001 |

| 103 | Bob Davis | 789 Oak Ave.| Chicago | IL | 60601 |

**Orders Table:**

| Order ID | Customer ID | Customer Name | Order Date | Total Amount |

|-------------|------------------|------------------------|----------------|------------------ -|

| 1001 | 101 | John Smith | 2023-06-15 | $100 |

| 1002 | 102 | Alice Johnson | 2023-06-16 | $150 |

| 1003 | 103 | Bob Davis | 2023-06-17 | $200 |

```

In this example, the denormalized schema allows us to retrieve the customer name directly from the `Orders` table without the need to perform a join operation with the `Customers` table. This can improve the performance of queries that involve retrieving order details along with customer information.

It's important to note that de-normalization introduces redundancy and requires careful management to ensure data consistency. Updates or changes to customer names need to be propagated correctly to maintain consistency across the duplicated attribute in the `Orders` table.

This example demonstrates how de-normalization can be applied to improve query performance by introducing redundancy and simplifying data retrieval. However, it's essential to carefully consider the trade-offs and assess the specific requirements of your application before deciding to de-normalize a database schema.