# DBMS Assignment

**Chethan Kumar KM**
**PES1PG23CA329**

**1. Write an example of a database table that is not normalized and explain how it can be normalized using Normalization Forms.**

Non-Normalized Table:

Consider a table Employee with the following columns:

| EmployeeID | EmployeeName | Department | DepartmentLocatio |
|---|---|---|---|
| 1 | Alice | IT | New York |
| 2 | Bob | HR | Chicago |
| 3 | Charlie | IT | San Francisco |
| 4 | David | HR | New York |

This table is not normalized because it contains repeating groups of data. For example, the department information (Department, DepartmentLocation) is repeated for each employee in the same department.

Normalized Tables using Normalization Forms:

First Normal Form (1NF):
To convert this table to 1NF, we need to remove repeating groups by creating a separate table for departments.

Employee Table:

| EmployeeID | EmployeeName | DepartmentID |
|---|---|---|
| 1 | Alice | 1 |
| 2 | Bob | 2 |
| 3 | Charlie | 1 |
| 4 | David | 2 |

Department Table:

| DepartmentID | Department | DepartmentLocation |
|---|---|---|
| 1 | IT | New York |
| 2 | HR | Chicago |
| 3 | IT | San Francisco |

Second Normal Form (2NF):
In 2NF, all attributes must be fully functional dependent on the primary key. In this case, both tables are already in 2NF because the Department table's attributes are fully dependent on the DepartmentID (primary key).
Third Normal Form (3NF):
In 3NF, there should be no transitive dependencies. In the Employee table, DepartmentID is a foreign key that determines Department, which is dependent on DepartmentID. This is not a transitive dependency, so the tables are in 3NF.
By normalizing the original non-normalized table, we have eliminated the repeating groups and ensured that each table represents a single entity (Employee or Department) without redundancy, making the database more efficient and easier to maintain.

**2. How does normalization contribute to efficient querying and data manipulation?**

Normalization contributes to efficient querying and data manipulation in several ways:

1. **Reduced Redundancy:** Normalization reduces data redundancy by organizing data into separate tables and linking them through relationships. This means that each piece of information is stored in only one place, reducing the risk of inconsistencies and making updates easier and faster.

2. **Improved Data Integrity:** By eliminating redundant data, normalization helps maintain data integrity. Updates, inserts, and deletes are less error-prone because there is only one place to make changes for each piece of data.

3. **Better Performance:** Normalization can lead to better performance for certain types of queries. Since related data is stored in separate tables, queries that only need information from one table can be faster and more efficient.

4. **Simplified Data Model:** A normalized database tends to have a simpler and more logical data model, making it easier for developers to understand and work with the database structure. This can lead to faster development times and easier maintenance.

5. **Scalability:** Normalization can improve the scalability of a database. As the database grows and more data is added, normalization helps maintain a consistent and manageable database structure, which can be important for performance and maintenance as the database grows.

Overall, normalization contributes to efficient querying and data manipulation by reducing redundancy, improving data integrity, providing better performance for certain queries, simplifying the data model, and enhancing scalability.

### 3. Explain the concept of functional dependencies in normalization

Functional dependencies are a key concept in normalization, defining the relationships between attributes in a table. In simple terms, a functional dependency exists when the value of one attribute uniquely determines the value of another attribute in the same table.

Let's consider a table Employee with the following attributes:

| EmployeeID | EmployeeName | Department | Salary |
|---|---|---|---|
| 1 | Alice | IT | 5000 |
| 2 | Bob | HR | 4500 |
| 3 | Charlie | IT | 5200 |
| 4 | David | HR | 4800 |

In this table, we can say that EmployeeID uniquely determines EmployeeName, Department, and Salary. This is because each EmployeeID corresponds to only one EmployeeName, Department, and Salary value. Therefore, we can say that EmployeeID → EmployeeName, EmployeeID → Department, and EmployeeID → Salary are functional dependencies.

Another example could be the Department attribute determining the DepartmentLocation. If each department is located in only one location, then Department → DepartmentLocation is a functional dependency.

Functional dependencies play a crucial role in normalization because they help identify redundant data and ensure that the database is structured efficiently. By understanding the functional dependencies in a table, we can normalize the table to eliminate redundancy and improve data integrity.

**4. How does indexing enhance the performance of database tables?**

- Faster Data Retrieval: When you create an index on a column in a database table, the database creates a data structure (often a B-tree or hash table) that stores the values of that column along with pointers to the corresponding rows in the table. This allows the database to quickly locate the rows that match a given value or range of values in the indexed column.

- Reduced Disk I/O: Indexing reduces the number of disk I/O operations required to retrieve data. Instead of scanning the entire table to find matching rows, the database can use the index to locate the relevant rows more efficiently. This can significantly reduce the time it takes to execute queries, especially for tables with a large number of rows.

- Improved Query Performance: Queries that involve indexed columns can perform much faster than equivalent queries that do not use indexes. For example, a query that selects rows based on a specific value in an indexed column can be executed much more quickly because the database can use the index to quickly locate the relevant rows.

- Support for Sorted Output: In some cases, indexes can also improve the performance of queries that require sorted output. By using an index that is sorted on the desired column, the database can avoid the need to sort the entire result set, which can be a costly operation for large data sets.

- Concurrency Control: Indexes can also improve concurrency control by reducing the time that locks are held on data. When a query uses an index to quickly locate rows, it can reduce the time that other transactions need to wait to access the same data.

Overall, indexing enhances the performance of database tables by speeding up data retrieval, reducing disk I/O, improving query performance, supporting sorted output, and enhancing concurrency control.

**5. What are the different ways to implement an UPDATE operation in SQL?**

In SQL, there are several ways to implement an UPDATE operation, depending on the specific requirements of the update and the database management system (DBMS) being used. Here are some common ways to perform an UPDATE operation in SQL:

**Simple UPDATE:** The most basic form of the UPDATE statement updates one or more columns in a table for all rows that meet a specified condition. The syntax is as follows:

```
UPDATE

SET column1 = value1, column2 = value2, ...
WHERE condition;
```

For example, to update the salary of an employee with EmployeeID 1:

```
UPDATE Employees
SET Salary = 60000
WHERE EmployeeID = 1;
```

**UPDATE with Subquery**: You can use a subquery to update values based on the result of another query. For example, to update the salary of all employees in the IT department to a certain value:

```
UPDATE Employees
SET Salary = 55000
WHERE Department = 'IT';
```

**UPDATE from Another Table**: You can update values in a table based on values from another table using a JOIN. For example, to update the salary of employees based on a mapping table:

```
UPDATE Employees
SET Salary = NewSalaries.NewSalary
FROM Employees
JOIN NewSalaries ON Employees.EmployeeID =
NewSalaries.EmployeeID;
```

**Conditional UPDATE:** You can use conditional logic within the UPDATE statement to update values based on different conditions. For example, to update the salary of employees based on their performance rating:

```
UPDATE Employees
SET Salary = CASE
WHEN PerformanceRating = 'Excellent' THEN Salary * 1.1
WHEN PerformanceRating = 'Good' THEN Salary * 1.05
ELSE Salary
END;
```

**Update with EXISTS**: You can use the EXISTS condition to update rows in one table based on the existence of corresponding rows in another table. For example, to update the status of orders that have corresponding entries in the OrderDetails table:

```
UPDATE Orders
SET Status = 'Shipped'
WHERE EXISTS (
SELECT *
FROM OrderDetails
WHERE OrderDetails.OrderID = Orders.OrderID
 );
```

## 6. Differentiate between delete and truncate operations with respect to TCL command?

Here's a comparison of the DELETE and TRUNCATE operations in SQL with respect to the Transaction Control Language (TCL) commands:

| DELETE | TRUNCATE |
|---|---|
| Removes rows based on a condition or all rows in a table. | Removes all rows from a table. |
| Generates an entry in the transaction log for each deleted row, allowing for rollback. | Not logged as individual row deletions, so cannot be rolled back (in most DBMS). |
| Typically locks each row being deleted, can cause blocking in concurrent environments. | Obtains a table-level lock, usually faster and causes less contention. |
| Deletes can violate referential integrity constraints, depending on the configuration. | Truncate cannot be used if there are foreign key constraints referencing the table. |
| Does not reset identity columns (e.g., auto-increment values). | Resets identity columns to their seed value (typically 1). |
| Slower for large tables due to logging and individual row deletions. | Faster for large tables as it is a bulk operation. |
| Can have a WHERE clause to delete specific rows based on a condition. | Cannot have a WHERE clause, deletes all rows in the table. |

Both DELETE and TRUNCATE are important operations in SQL, but they are used in different scenarios based on the requirements and constraints of the database system.