Data Normalisation

Data normalisation is a process that is used to remove the inconsistency and redundancy present inside the data of a table. These inconsistencies are referred to as anomalies and the steps to remove these anomalies are as follows:

- Step 1: Converting a given table to the **1st Normal Form (1NF)**
- Step 2: Converting the table from 1NF to the **2nd Normal Form (2NF)**
- Step 3: Converting the table from 2NF to the **3rd Normal Form (3NF)**

Anomalies in a Database

Anomalies in data can occur irrespective of whether the data is modelled properly or not. These anomalies are of the following types:

- **Insertion anomaly**: It occurs when you insert a data record as a row but the information is not available for all the columns, causing null values in a row.
- **Updation anomaly**: It occurs when you update information in one row of a table, causing the previous information to be deleted, as there is no other copy of that data.
- **Deletion anomaly**: It occurs when deletion of one row to remove data from some columns causes the data in other columns to be deleted as well. This occurs when data about multiple entities are stored in the same table. If information about one entity is deleted, then the information about another entity in the same row gets deleted.

1st Normal Form (1NF)

The first step of normalisation is to convert a table to 1NF. This implies that every field of a table will contain single values and every table will have a primary key.

Consider the following table.

Customer ID	**Customer Name**	**Car Number Plate**	**Car Name**	**Date of Transaction**	**Owner ID**	**Owner Name**
C12	Sachin	CarQ1234	Swift	12/01/2020	O76	Dev
C12	Sachin	CarQ5436	Thar	18/01/2020	O54	Rohit
C46	Rahul	CarQ3421	Baleno	12/01/2020	O54	Rohit
C46	Rahul	CarQ6534	Honda City	14/01/2020	O65	Shikhar
C46	Rahul	CarQ3789	Swift	15/01/2020	O86	Irfan

We have already separated multiple values into different rows. Each row in this table contains a unique transaction. To identify each row, you must find a primary key for this table.

Single attribute keys:

- Customer ID: One customer can rent many cars.
- Car Number Plate: One car can be rented multiple times.
- Date of Transaction: Many transactions can occur in one day.

Multiple attribute keys:

- Customer ID and Car Number Plate: One customer can rent many cars.
- Customer ID and Date of Transaction: One customer can rent two cars on the same day.
- Car Number Plate and Date of Transaction: One car can be rented only once a day.

Car Number Plate and Date of Transaction are unique for each row. They together form the composite key for this table.

2nd Normal Form (2NF)

The **second normal form** removes partial functional dependencies in the table. For a table to be in 2NF, it must be in 1NF. Let’s understand this concept in more detail.

Suppose two columns, column B and column C, form a composite key for a table. The value of column B can be used to determine the value of column D. The value of column C can be used to determine the values of columns E and F. The values of columns B and C together can be used to determine the values of columns A, D, E and F.

- (B, C) -> A, D, E, F
- B -> D
- C -> E, F

Column B and column C are the prime attributes here. The values of columns D, E and F depend on a prime attribute and not on the entire composite key. This is a case of partial dependency on the composite key. The value of column A is dependent on the values of both columns B and C. This is a case of full functional dependency on the composite key.

To remove partial dependencies, you need to separate the partial dependencies into new tables. In this case, the following three tables will be formed:

1. Table 1: <A, B, C>. Column B and column C together will act as a composite key and column A is fully functionally dependent on the composite key.
2. Table 2: <B, D>. Column B will act as a primary key and column D is dependent on this primary key.
3. Table 3: <C, E, F>. Column C will act as a primary key. Column E and column F will be dependent on this primary key.

3rd Normal Form (3NF)

For a table to be in 3NF, it must be in 2NF and there should be no transitive dependencies. Consider a table with the following three attributes: <A, B, C>. Consider that A is the primary key. A transitive dependency occurs when a non-prime attribute C is dependent on another non-prime attribute B, which depends on the prime attribute A. In this case, you create a new table with columns B and C. Column B acts as a primary key for this new table. Then, the two resultant tables are as follows:
<A, B> and <B, C>.

Thus, the conditions for a table to be in 3NF are as follows:

1. Each field of the table must have single values. The table must be in 1NF.
2. Every non-prime attribute must be fully functionally dependent on the composite key. The table must be in 2NF.
3. The table must have no transitive dependencies.

Query Optimisation and Best Practices

So far, you have learnt how to use different commands and tools in SQL. There is, however, one marked difference between the data sets that we used and the ones that are used by the likes of Amazon and Uber. The difference, of course, is the size of data.

In this course, you saw a small database for simplicity and ease of understanding. However, real-life data sets, such as those used by Amazon and Uber, would have millions, or even billions, of rows. Hence, although a 'select' operation on our dummy tables takes less than a fraction of a second, a similar query, when applied on a table with millions of rows, for example, a 'customer' or a 'product' table for Amazon, might take hours if not optimised properly.

Thus comes the need for query optimisation and the various tips and techniques used in order to save the runtime and the required computational power.

Best Practices

Suppose you want to buy a new car. Now, once you enter a showroom, a salesperson will explain the features of various cars to you, including each model's cost, mileage, etc. They will probably also mention an optimum range of speed that you should maintain while driving. But why is this information important? Maintaining this range of speed ensures that the car continues to give the best possible mileage.

Similarly, you should follow best practices while writing any SQL code. This makes your code more readable for the people working on it and helps maximise the output for your company.

Some of the important best practices are as follows:

- Comment your code by using a hyphen **(-)** for a single line and **(/* ... */)** for multiple lines of code.
- Always use table aliases when your query involves more than one source table.
- Assign simple and descriptive names to columns and tables.
- Write SQL keywords in upper case and the names of columns, tables and variables in lower case.
- Always use column names in the 'order by' clause, instead of numbers.
- Maintain the right indentation for different sections of a query.
- Use new lines for different sections of a query.
- Use a new line for each column name.
- Use the SQL Formatter or the MySQL Workbench Beautification tool (Ctrl + B).

Indexing

Indexing is the process of referring to only the required value(s) directly, instead of going through the entire table. This prevents the query engine from looking up values one by one; instead, it returns the exact value that you want right away. Indexing is necessary for querying extremely large data sets. A primary key is an index because it helps identify each record in a table uniquely. You cannot actually see an index, as it is an internal construct used in database engines to speed up queries.

MySQL has a specific DDL statement called CREATE INDEX, which is used to specify the attribute on which you want to create an index. Generally, you would not have the permission to create such indices on a database. In such cases, you can ask the Database Administrator (DBA) to create the index for you.

The command for creating an index is as follows:
CREATE INDEX index_name
ON table_name (column_1, column_2, ...);

The command for adding an index is as follows:
ALTER TABLE table_name
ADD INDEX index_name(column_1, column_2, ...);

The command for dropping an index is as follows:
ALTER TABLE table_name
DROP INDEX index_name;

Clustered vs Non-Clustered Indexing

There are two types of indices: **clustered** and **non-clustered**. The differences between these two types of indices are listed in the table below.

Clustered Index	**Non-Clustered Index**

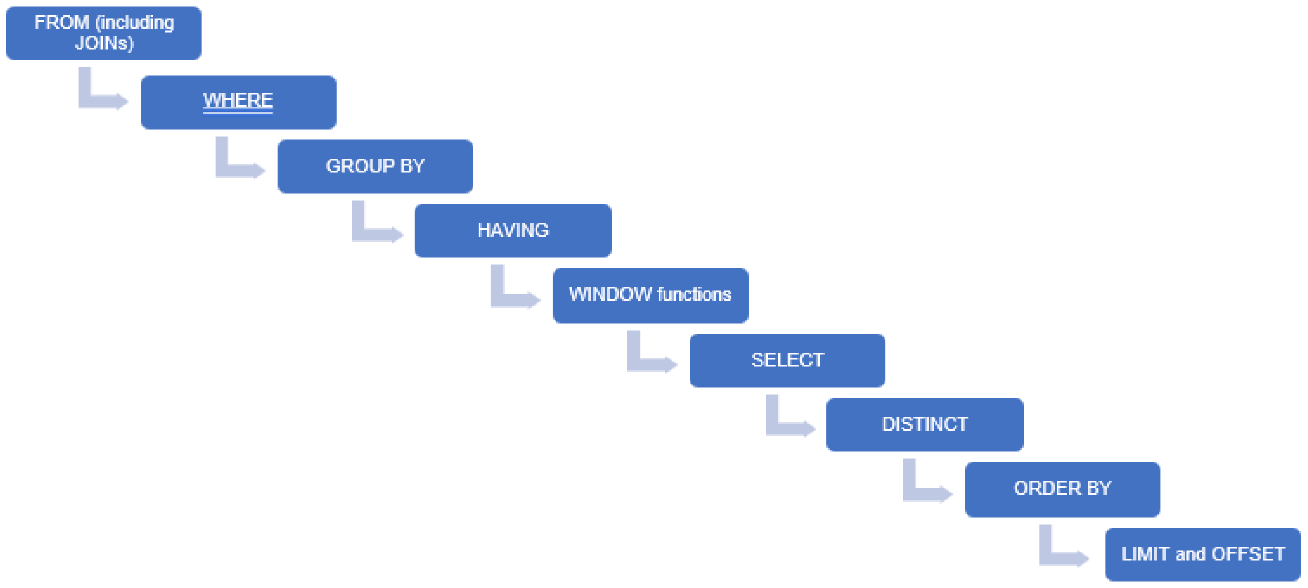
1. This is mostly used as the primary key of the table.	1. This is a combination of one or more columns of the table.
2. It is present within the table.	2. The unique list of keys is present outside the table.
3. It does not require a separate mapping.	3. The external table points to different sections of the main table.
4. It is relatively faster.	4. It is relatively slower.

Order of Query Execution

The order in which the various SQL statements appear in a query is as follows:

1. SELECT
2. FROM
3. [JOIN]
4. WHERE
5. GROUP BY
6. HAVING
7. WINDOW
8. ORDER BY

However, the order in which the various statements are executed by the database engine is not the same. This order is depicted in the following diagram.



Some of the important points that you should keep in mind while writing a query are as follows:

- Use inner joins wherever possible to avoid having any unnecessary rows in the resultant table.
- Apply all the required filters to get only the required data values from multiple tables.
- Index the columns that are frequently used in the WHERE clause.

- Avoid using DISTINCT while using the GROUP BY clause, as it slows down query processing.
- Avoid using SELECT * as much as possible. Select only the required columns.
- Use the ORDER BY clause only if it is absolutely necessary, as it is processed late in a query.
- Avoid using LIMIT and OFFSET as much as possible. Instead, apply appropriate filters using the WHERE clause.

Joins vs Nested Queries

As you learnt previously, nested queries and joins are used to retrieve data from multiple tables. However, is one of them more efficient than the other, especially in the case of large data sets? Well, it depends on the query processor. Query processors run optimising operations on your queries to ensure that the runtime is as low as possible.

Executing a statement with the 'join' clause creates a join index, which is an internal indexing structure. This makes it more efficient than a nested query. However, a nested query would perform better than a join while querying data from a distributed database.

In a distributed database, tables are stored in different locations instead of a local system. In this case, a nested query would perform better than a join, as you can extract relevant information from different tables located in different computers. You can then merge the values in order to obtain the desired result. In the case of a join, you would need to create a large table from the existing tables and filtering this large table would require comparatively more time.

Case Statements, Stored-Routines and Cursors

Case Statements

Case statements are used to classify your results based on certain conditions. Suppose you are in school and have to follow a set time-table, according to which you need to bring the textbooks of certain subjects on certain days to class. Consider the following time-table:

- Monday: English and Hindi
- Tuesday: English and Maths
- Wednesday: Science and Social Science
- Thursday: Maths and Social Science
- Friday: Hindi and Science

This is a basic example where you can determine the subjects that you would be studying in class on certain days by just looking at the list. However, the data, in general, is enormous and complex. You may need to set many conditions and perform many actions according to each of these conditions.

The syntax for writing a case statement is as follows:

CASE

```
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  .
  .
  WHEN conditionN THEN resultN
  ELSE result
END AS column_name;
```

User-Defined Functions (UDFs)

You already know how to deploy various in-built MySQL functions, such as sum(), avg() and concat(), to make querying easier. Now, it is possible to find the sum of two numbers in MySQL without using the sum() function. You can use the arithmetic addition operator (+) for this purpose. Similarly, you can use the arithmetic addition and division operators to find the average of two numbers.

The sum() and avg() functions are preferred because they are easier to use and also increase the readability of the code. Another important factor is reusability. You do not need to see the entire definition behind the sum() function every time you use it; all you need is the name ‘sum()’ to invoke the function whenever you want to determine the sum of two numbers.

However, you may want to repeat some operations multiple times in a piece of code because they do not have an in-built function. This is where you must use user-defined functions (UDFs) or stored functions.

The syntax for writing a UDF is as follows:

```
DELIMITER $$

CREATE FUNCTION function_name(func_parameter1, func_parameter2, ...)
  RETURN datatype [characteristics]
/*      func_body      */
  BEGIN
    <SQL Statements>
    RETURN expression;
END $$

DELIMITER ;
```

To call this function, you can use the following syntax:

```
CALL function_name(func_parameter1, func_parameter2, ...);
```

Remember the following points:

- The CREATE FUNCTION is also a DDL statement.
- The function body must contain one RETURN statement.

Stored Procedures

A stored procedure is similar to a UDF but differs in certain ways.

The syntax for writing a stored procedure is as follows:

```
DELIMITER $$

CREATE PROCEDURE Procedure_name (<Parameter List>)
BEGIN
  <SQL Statements>
END $$

DELIMITER ;
```

To call this stored procedure, you can use the following syntax:

```
CALL Procedure_name(<Parameter List>);
```

Although the syntax of a stored procedure is similar to that of UDFs, there are many differences between these two. The differences between UDFs and stored procedures are listed in the table below.

UDF	**Stored Procedure**
1. It supports only the input parameter, not the output.	1. It supports input, output and input-output parameters.
2. It cannot call a stored procedure.	2. It can call a UDF.
3. It can be called using any SELECT statement.	3. It can be called using only a CALL statement.
4. It must return a value.	4. It need not return a value.
5. Only the 'select' operation is allowed.	5. All database operations are allowed.

Cursors

While stored routines are a great way to store and reuse logic in code as required, they do have some disadvantages, such as they are not portable across different database engines. Cursors are used to individually process each row that is returned in a query.