1. What is anomalies? Explain Update anomaly in SQL.

Anomaly means inconsistency in the pattern from the normal form. In Database Management System (DBMS), anomaly means the inconsistency occurred in the relational table during the operations performed on the relational table.

There can be various reasons for anomalies to occur in the database. For example, if there is a lot of redundant data present in our database then DBMS anomalies can occur. If a table is constructed in a very poor manner then there is a chance of database anomaly. Due to database anomalies, the integrity of the database suffers.

The other reason for the database anomalies is that all the data is stored in a single table. So, to remove the anomalies of the database, normalization is the process which is done where the splitting of the table and joining of the table (different types of join) occurs.

We will see the anomalies present in a table by the different examples:

Example 1:

Worker_id	Worker_name	Worker_dept	Worker_address
65	Ramesh	ECT001	Jaipur
65	Ramesh	ECT002	Jaipur
73	Amit	ECT002	Delhi
76	Vikas	ECT501	Pune
76	Vikas	ECT502	Pune
79	Rajesh	ECT669	Mumbai

In the above table, we have four columns which describe the details about the workers like their name, address, department and their id. The above table is not normalized, and there is definitely a chance of anomalies present in the table.

There can be three types of an anomaly in the database:

Updation / Update Anomaly

When we update some rows in the table, and if it leads to the inconsistency of the table then this anomaly occurs. This type of anomaly is known as an updation anomaly. In the above table, if we want to update the address of Ramesh then we will have to update all the rows where Ramesh is present. If during the update we miss any single row, then there will be two addresses of Ramesh, which will lead to inconsistent and wrong databases.

Insertion Anomaly

If there is a new row inserted in the table and it creates the inconsistency in the table then it is called the insertion anomaly. For example, if in the above table, we create a new row of a worker, and if it is not allocated to any department then we cannot insert it in the table so, it will create an insertion anomaly.

Deletion Anomaly

If we delete some rows from the table and if any other information or data which is required is also deleted from the database, this is called the deletion anomaly in the database. For example, in the above table, if we want to delete the department number ECT669 then the details of Rajesh will also be deleted since Rajesh's details are dependent on the row of ECT669. So, there will be deletion anomalies in the table.

To remove this type of anomalies, we will normalize the table or split the table or join the tables. There can be various normalized forms of a table like 1NF, 2NF, 3NF, BCNF etc. we will apply the different normalization schemes according to the current form of the table.

Example 2:

Stu_id	Stu_name	Stu_branch	Stu_club
2018nk01	Shivani	Computer science	literature
2018nk01	Shivani	Computer science	dancing
2018nk02	Ayush	Electronics	Videography
2018nk03	Mansi	Electrical	dancing
2018nk03	Mansi	Electrical	singing
2018nk04	Gopal	Mechanical	Photography

In the above table, we have listed students with their name, id, branch and their respective clubs.

Updation / Update Anomaly

In the above table, if Shivani changes her branch from Computer Science to Electronics, then we will have to update all the rows. If we miss any row, then Shivani will have more than one branch, which will create the update anomaly in the table.

Insertion Anomaly

If we add a new row for student Ankit who is not a part of any club, we cannot insert the row into the table as we cannot insert null in the column of stu_club. This is called insertion anomaly.

Deletion Anomaly

If we remove the photography club from the college, then we will have to delete its row from the table. But it will also delete the table of Gopal and his details. So, this is called deletion anomaly and it will make the database inconsistent.

2. What is Normalization? State the types of Normalization.

Normalization in DBMS refers to the process of organizing data in a database into separate tables in order to reduce data redundancy and improve data integrity. The goal of normalization is to eliminate data anomalies that can lead to data inconsistencies, and to make sure that data is stored in a logical, consistent manner. There are several normalization forms (such as First Normal Form, Second Normal Form, Third Normal Form, etc.) that provide guidelines for organizing data in a normalized way. Normalization is an important aspect of database design and can help to improve the performance and scalability of a database.

Types of Normalization

There are several types of normalization in DBMS, each with its own set of rules and guidelines for organizing data in a normalized way. The most commonly used normalization forms include:

1. **First Normal Form (1NF):** All data must be atomic, meaning that each cell in a table should contain only a single value and not a list of values.
2. **Second Normal Form (2NF):** In addition to meeting the rules of 1NF, a table must not contain any partial dependencies. A partial dependency

exists when a non-primary key column depends on only part of a composite primary key.

3. **Third Normal Form (3NF):** In addition to meeting the rules of 2NF, a table must not contain any transitive dependencies. A transitive dependency exists when a non-primary key column depends on another non-primary key column.
4. **Boyce-Codd Normal Form (BCNF):** A relation is in BCNF if and only if for every one of its non-trivial functional dependencies $X \rightarrow Y$, X is a superkey.
5. **Fourth Normal Form (4NF):** A table is in 4NF if it is in BCNF and it has no multi-valued dependencies.
6. **Fifth Normal Form (5NF):** A relation is in 5NF if every non-trivial join dependency in R is implied by the candidate keys of R.

First Normal Form (1NF)

First Normal Form (1NF) is the most basic level of normalization in a DBMS. The rules for achieving 1NF are as follows:

1. Each table should have a primary key, which uniquely identifies each record in the table.
2. Each column in the table should contain only atomic values, which means that a single cell should contain a single value and not a list of values.
3. There should be no repeating groups of data.

For example, let's say we have a table named "Students" that stores information about students in a school. A table that is not in 1NF might look like this:

StudentID	Name	Subject	Grade
1	John Smith	Math, Science	A
2	Jane Doe	English, History	B

In this table, the Subject column contains multiple values separated by commas, which violates the rule of atomic values. This table is not in 1NF. To bring this table to 1NF, we would need to split the Subject column into multiple columns, one for each subject, and repeat the student information for each subject taken.

StudentID	Name	Subject	Grade
1	John Smith	Math	A
1	John Smith	Science	A
2	Jane Doe	English	B
2	Jane Doe	History	B

This table is now in 1NF, since all the data is atomic, each cell contains only one value, and there are no repeating groups of data.

1NF is the starting point for normalization, and to ensure the data integrity and consistency it's necessary to move to next normalization forms.

Second Normal Form (2NF)

Second Normal Form (2NF) builds upon the rules of First Normal Form (1NF) by addressing the issue of partial dependencies. In 2NF, a table must not have any partial dependencies. A partial dependency exists when a non-primary key column depends on only part of a composite primary key.

For example, let's say we have a table named "Orders" that stores information about customer orders. A table that is not in 2NF might look like this:

OrderID	CustomerID	Product	Quantity	Price
1	1	A	2	10
2	1	B	1	20
3	2	A	3	10

In this table, the Price column depends on the Product column and the primary key is composed of OrderID and CustomerID. The table is not in 2NF because the Price column is functionally dependent on only part of the primary key (Product) and not on the whole primary key (OrderID and CustomerID).

To bring this table to 2NF, we need to separate the table into two separate tables: one for the Orders and one for the Products.

Table: Orders

OrderID	CustomerID	Product	Quantity
1	1	A	2
2	1	B	1
3	2	A	3

Table: Products

Product	Price
A	10
B	20

Now, the Price column is dependent on the primary key of the Products table (Product) and the Orders table has no partial dependencies. This design is now in 2NF.

2NF eliminates partial dependencies and improves the data integrity by reducing the data anomalies. However, it's not enough to ensure the data consistency and to avoid data anomalies, so it's necessary to move to the next normalization forms.

Third Normal Form (3NF)

Third Normal Form (3NF) builds upon the rules of Second Normal Form (2NF) by addressing the issue of transitive dependencies. In 3NF, a table must not have any transitive dependencies. A transitive dependency exists when a non-primary key column depends on another non-primary key column, rather than on the primary key.

To achieve 3NF, a table must already be in 2NF and all non-primary key columns must be directly dependent on the primary key.

For example, let's say we have a table named "Employees" that stores information about employees in a company. A table that is not in 3NF might look like this:

EmployeeID	Name	Department	Manager
1	John	IT	Tom
2	Jane	HR	Tom
3	Tom	Management	

In this table, the Manager column depends on the Department column, and neither of them depend on the primary key (EmployeeID). This table is not in 3NF because of the transitive dependency between the Manager column and the Department column.

To bring this table to 3NF, we need to separate the table into two separate tables: one for the Employees and one for the Departments.

Table: Employees

EmployeeID	Name	Department
1	John	IT
2	Jane	HR
3	Tom	Management

Table: Departments

Department	Manager
IT	Tom
HR	Tom
Management	

Now, the Manager column is dependent on the primary key of the Departments table (Department) and the Employees table has no transitive dependencies. This design is now in 3NF.

3NF eliminates transitive dependencies and improves the data integrity and consistency by reducing the data anomalies. However, it's still not enough to ensure the data consistency and to avoid data anomalies, so it's necessary to move to the next normalization forms like Boyce-Codd Normal Form (BCNF) or Fourth Normal Form (4NF) in some cases.

Boyce-Codd Normal Form (BCNF)

Boyce-Codd Normal Form (BCNF) is a higher level of normalization than Third Normal Form (3NF). It addresses the issue of non-trivial functional dependencies. A functional dependency is said to be non-trivial if it doesn't hold true for the case where the left-hand side is a subset of the primary key.

A table is in BCNF if and only if for every non-trivial functional dependency X → Y, X is a superkey. A superkey is a set of columns that can uniquely identify a row in a table.

For example, let's say we have a table named "Invoices" that stores information about invoices of a company. A table that is not in BCNF might look like this:

InvoiceNumber	CustomerID	Product	Quantity	Price
1	1	A	2	10
2	1	B	1	20
3	2	A	3	10

In this table, the primary key is (InvoiceNumber) and it has a non-trivial functional dependency CustomerID → Product. This dependency doesn't hold true for the case where the left-hand side is a subset of the primary key (InvoiceNumber) because the product can be different for the same customer. So this table is not in BCNF.

To bring this table to BCNF, we need to separate the table into two separate tables: one for the Invoices and one for the Customer_Product.

Table: Invoices

InvoiceNumber	Quantity	Price
1	2	10
2	1	20
3	3	10

Table: Customer_Product

CustomerID	Product
1	A
1	B
2	A

Now, the table Invoices has the primary key (InvoiceNumber) and the table Customer_Product has the primary key (CustomerID, Product) . This design is now in BCNF.

BCNF is considered to be the strongest normal form and it eliminates non-trivial functional dependencies and improves the data integrity and consistency by reducing the data anomalies. However, it's not necessary to go through BCNF before going to higher normal forms like fourth normal form (4NF) or fifth normal form (5NF).

Fourth Normal Form (4NF)

Fourth Normal Form (4NF) is a higher level of normalization than Boyce-Codd Normal Form (BCNF). It addresses the issue of multi-valued dependencies. A multi-valued dependency is a type of dependency where a non-primary key column is functionally dependent on two or more independent non-primary key columns.

A table is in 4NF if and only if it does not contain any multi-valued dependencies.

For example, let's say we have a table named "Orders" that stores information about orders of a company. A table that is not in 4NF might look like this:

OrderNumber	Customer	Product	Supplier
1	John	A	S1
2	Jane	B	S2
3	Tom	A	S1

In this table, there is a multi-valued dependency between Product and Supplier. This dependency doesn't hold true for the case where a product is supplied by multiple suppliers or a supplier supplies multiple products. So this table is not in 4NF.

To bring this table to 4NF, we need to separate the table into two separate tables: one for the Orders and one for the Product_Supplier.

Table: Orders

OrderNumber	Customer	Product
1	John	A

2	Jane	B
3	Tom	A

Table: Product_Supplier

Product	Supplier
A	S1
B	S2

Now, the table Orders has no multi-valued dependencies and the table Product_Supplier has primary key (Product, Supplier) . This design is now in 4NF.

4NF eliminates multi-valued dependencies and improves the data integrity and consistency by reducing the data anomalies. However, it's not necessary to go through 4NF before going to higher normal forms like fifth normal form (5NF).

Fifth Normal Form (5NF)

Fifth Normal Form (5NF), also known as Projection-Join Normal Form (PJ/NF) is a higher level of normalization than Fourth Normal Form (4NF). It addresses the issue of join dependency. A join dependency is a type of dependency where a table can be split into two or more tables, each of which contains a subset of the original table's columns, and can be recombined by joining the tables on their common columns.

A table is in 5NF if and only if it does not contain any join dependencies.

For example, let's say we have a table named "Employee_Department" that stores information about employees and their departments. A table that is not in 5NF might look like this:

EmployeeID	EmployeeName	Department	Salary
1	John	IT	50000
2	Jane	HR	40000
3	Tom	IT	60000

In this table, there is a join dependency between Employee and Department. This dependency doesn't hold true for the case where an employee can belong to multiple departments or a department can have multiple employees. So this table is not in 5NF.

To bring this table to 5NF, we need to separate the table into three separate tables: one for the Employee, one for the Department, and one for the Employee_Department_Salary.

Table: Employee

EmployeeID	EmployeeName
1	John
2	Jane
3	Tom

Table: Department

Department
IT
HR

Table: Employee_Department_Salary

EmployeeID	Department	Salary
1	IT	50000
2	HR	40000
3	IT	60000

Now, the tables Employee, Department, and Employee_Department_Salary has no join dependency and each table has its own primary key. This design is now in 5NF.

5NF eliminates join dependencies and improves the data integrity and consistency by reducing the data anomalies. However, it’s not often used in practice as it requires a large number of tables and can be difficult to maintain.