**Explain the different steps in DB design**

The design process consists of the following steps:

* **Determine the purpose of your database**

This helps prepare you for the remaining steps.

* **Find and organize the information required**

Gather all of the types of information you might want to record in the database, such as product name and order number.

* **Divide the information into tables**

Divide your information items into major entities or subjects, such as Products or Orders. Each subject then becomes a table.

* **Turn information items into columns**

Decide what information you want to store in each table. Each item becomes a field, and is displayed as a column in the table. For example, an Employees table might include fields such as Last Name and Hire Date.

* **Specify primary keys**

Choose each table’s primary key. The primary key is a column that is used to uniquely identify each row. An example might be Product ID or Order ID.

* **Set up the table relationships**

Look at each table and decide how the data in one table is related to the data in other tables. Add fields to tables or create new tables to clarify the relationships, as necessary.

* **Refine your design**

Analyze your design for errors. Create the tables and add a few records of sample data. See if you can get the results you want from your tables. Make adjustments to the design, as needed.

* **Apply the normalization rules**

Apply the data normalization rules to see if your tables are structured correctly. Make adjustments to the tables, as needed.

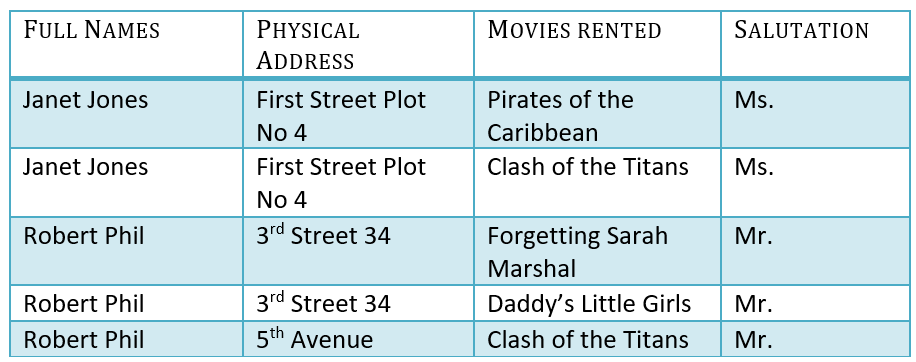
**What is normalization? Why do we need to do normalization? Explain 1NF, 2NF, 3NF with example**

**NORMALIZATION** is a database design technique that organizes tables in a manner that reduces redundancy and dependency of data. Normalization divides larger tables into smaller tables and links them using relationships. The purpose of Normalization is to eliminate redundant (useless) data and ensure data is stored logically.

The inventor of the relational model Edgar Codd proposed the theory of normalization with the introduction of the First Normal Form, and he continued to extend theory with Second and Third Normal Form. Later he joined Raymond F. Boyce to develop the theory of Boyce-Codd Normal Form.

## ****1NF (First Normal Form) Rules****

* Each table cell should contain a single value.
* Each record needs to be unique.

[[](https://www.guru99.com/images/1NF.png)](https://www.guru99.com/images/1NF.png)

## 2NF (Second Normal Form) Rules

* Be in 1NF
* Single Column Primary Key

It is clear that we can't move forward to make our simple database in 2nd Normalization form unless we partition the table above.

[[](https://www.guru99.com/images/Table2.png)](https://www.guru99.com/images/Table2.png)

[[](https://www.guru99.com/images/Table1.png)](https://www.guru99.com/images/Table1.png)

We have divided our 1NF table into two tables viz. Table 1 and Table2. Table 1 contains member information. Table 2 contains information on movies rented.

We have introduced a new column called Membership\_id which is the primary key for table 1. Records can be uniquely identified in Table 1 using membership id.

## 3NF (Third Normal Form) Rules

* Be in 2NF
* Has no transitive functional dependencies

To move our 2NF table into 3NF, we again need to again divide our table.

### 3NF Example

[[](https://www.guru99.com/images/2NFTable1.png)](https://www.guru99.com/images/2NFTable1.png)

[[](https://www.guru99.com/images/2NFTable2.png)](https://www.guru99.com/images/2NFTable2.png)

[[](https://www.guru99.com/images/2NFTable3.png)](https://www.guru99.com/images/2NFTable3.png)

We have again divided our tables and created a new table which stores Salutations.

There are no transitive functional dependencies, and hence our table is in 3NF

In Table 3 Salutation ID is primary key, and in Table 1 Salutation ID is foreign to primary key in Table 3

**When do we have to do denormalization of DB?**

Denormalization is a strategy used on a previously-normalized database to increase performance. The idea behind it is to add redundant data where we think it will help us the most. We can use extra attributes in an existing table, add new tables, or even create instances of existing tables.

**Explain the different types of joins available in RDBMS**

We can classify joins basically into two types

1. **INNER JOINS**: These joins are the one that has the tuples that satisfy some conditions and rest are discarded . Further they are classified as

* Theta join
* Equi join
* Natural join

1. **OUTER JOINS**: These have all the tuples from either or both the relations. Further they are classified as

* Left outer join
* Right outer join
* Full outer join

### ****INNER JOINS****

1. **Theta join(θ)** – They have tuples from different relations if and only if they satisfy the theta condition, here the comparison operators (≤, ≥, ˂, ˃, =, ̚ )come into picture. Let us consider simple example to understand in a much better way, suppose we want to buy a mobile and a laptop, based on our budget we have thought of buying both such that mobile price should be less than that of laptop.

Consider the following tables.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Table A** | |  | **Table B** | | |
| **column 1** | **column 2** | |  | **column 1** | **column 2** | |
| 1 | 1 | |  | 1 | 1 | |
| 1 | 2 | |  | 1 | 3 | |

For example:

A ⋈ A.column 2 > B.column 2 (B)

|  |  |
| --- | --- |
| **A ⋈ A.column 2 > B.column 2 (B)** | |
| **column** | **column 2** |
| 1 | 2 |

1. **EQUI Join**

**EQUI JOIN** is done when a Theta join uses only the equivalence condition. EQUI join is the most difficult operation to implement efficiently in an RDBMS, and one reason why RDBMS have essential performance problems.

For example:

A ⋈ A.column 2 = B.column 2 (B)

|  |  |
| --- | --- |
| **A ⋈ A.column 2 = B.column 2 (B)** | |
| **column 1** | **column 2** |
| 1 | 1 |

1. **NATURAL JOIN** does not utilize any of the comparison operators. In this type of join, the attributes should have the same name and domain. In Natural Join, there should be at least one common attribute between two relations.

It performs selection forming equality on those attributes which appear in both relations and eliminates the duplicate attributes.

Example:

Consider the following two tables

|  |  |
| --- | --- |
| **C** | |
| **Num** | **Square** |
| 2 | 4 |
| 3 | 9 |

|  |  |
| --- | --- |
| **D** | |
| **Num** | **Cube** |
| 2 | 8 |
| 3 | 18 |

C ⋈ D

|  |  |  |
| --- | --- | --- |
| **C ⋈ D** | | |
| **Num** | **Square** | **Cube** |
| 2 | 4 | 8 |
| 3 | 9 | 18 |

**OUTER JOINS:-**

**Left Outer Join (A [https://www.guru99.com/images/1/100518_0535_RelationalA5.png](https://www.guru99.com/images/1/100518_0535_RelationalA5.png) B)**

**LEFT JOIN** returns all the rows from the table on the left even if no matching rows have been found in the table on the right. When no matching record found in the table on the right, NULL is returned.

[](https://www.guru99.com/images/1/100518_0535_RelationalA6.png)

Consider the following 2 Tables

|  |  |
| --- | --- |
| **A** | |
| **Num** | **Square** |
| 2 | 4 |
| 3 | 9 |
| 4 | 16 |

|  |  |
| --- | --- |
| **B** | |
| **Num** | **Cube** |
| 2 | 8 |
| 3 | 18 |
| 5 | 75 |

A [https://www.guru99.com/images/1/100518_0535_RelationalA5.png](https://www.guru99.com/images/1/100518_0535_RelationalA5.png) B

|  |  |  |
| --- | --- | --- |
| **A ⋈ B** | | |
| **Num** | **Square** | **Cube** |
| 2 | 4 | 8 |
| 3 | 9 | 18 |
| 4 | 16 | - |

**Right Outer Join ( A [https://www.guru99.com/images/1/100518_0535_RelationalA7.png](https://www.guru99.com/images/1/100518_0535_RelationalA7.png) B )**

**RIGHT JOIN** returns all the columns from the table on the right even if no matching rows have been found in the table on the left. Where no matches have been found in the table on the left, NULL is returned. RIGHT outer JOIN is the opposite of LEFT JOIN

In our example, let's assume that you need to get the names of members and movies rented by them. Now we have a new member who has not rented any movie yet.

[](https://www.guru99.com/images/1/100518_0535_RelationalA8.png)

A [https://www.guru99.com/images/1/100518_0535_RelationalA7.png](https://www.guru99.com/images/1/100518_0535_RelationalA7.png) B

|  |  |  |
| --- | --- | --- |
| **A ⋈ B** | | |
| **Num** | **Cube** | **Square** |
| 2 | 8 | 4 |
| 3 | 18 | 9 |
| 5 | 75 | - |

**Full Outer Join ( A [https://www.guru99.com/images/1/100518_0535_RelationalA9.png](https://www.guru99.com/images/1/100518_0535_RelationalA9.png) B)**

In a **FULL OUTER JOIN** , all tuples from both relations are included in the result, irrespective of the matching condition.

Example:

A [https://www.guru99.com/images/1/100518_0535_RelationalA9.png](https://www.guru99.com/images/1/100518_0535_RelationalA9.png) B

|  |  |  |
| --- | --- | --- |
| **A ⋈ B** | | |
| **Num** | **Square** | **Cube** |
| 2 | 4 | 8 |
| 3 | 9 | 18 |
| 4 | 16 | - |
| 5 | - | 75 |

**What is primary key, foreign key?**

**PRIMARY KEY:-** A table can have just a single primary key, which may comprise of single or different fields. At the point when numerous fields are utilized as a primary key, they are known as a composite key.  
In the event that a table has a primary key characterized on any field(s), at that point you can’t have two records having a similar estimation of that field(s).

* Create Primary Key SQL Server

The following SQL Primary Key example sets ID as a primary key in a CUSTOMERS table.

1. Make TABLE **CUSTOMERS**(
2. ID INT NOT NULL,
3. NAME **VARCHAR** (20) NOT NULL,
4. AGE INT NOT NULL,
5. ADDRESS **CHAR** (25) ,
6. Salary **DECIMAL** (18, 2),
7. PRIMARY **KEY** (ID)

**FOREIGN KEY:-** A Foreign Key is a segment or a mix of segments whose qualities coordinate a Primary Key in an alternate table.  
The connection between 2 tables coordinates the Primary Key in one of the tables with a Foreign Key in the second table.  
Example of SQL Foreign Key –  
**CUSTOMERS table**

1. CREATE TABLE **CUSTOMERS**(
2. ID INT NOT NULL,
3. NAME **VARCHAR** (20) NOT NULL,
4. AGE INT NOT NULL,
5. ADDRESS **CHAR** (25) ,
6. SALARY **DECIMAL** (18, 2),
7. PRIMARY **KEY** (ID)
8. );

**ORDERS table**

1. CREATE TABLE **ORDERS** (
2. ID INT NOT NULL,
3. DATE DATETIME,
4. CUSTOMER\_ID INT references **CUSTOMERS**(ID),
5. AMOUNT double,
6. PRIMARY **KEY** (ID)
7. );

**What is a trigger?**

A database trigger is special [stored procedure](https://www.essentialsql.com/what-is-a-stored-procedure/) that is run when specific actions occur within a database.  Most triggers are defined to run when changes are made to a table’s data.  Triggers can be defined to run *instead of* or *after* DML (Data Manipulation Language) actions such as INSERT, UPDATE, and DELETE.

Triggers help the database designer ensure certain actions, such as maintaining an audit file, are completed regardless of which program or user makes changes to the data.

The programs are called triggers since an event, such as adding a record to a table, fires their execution.

**What is the difference between a stored procedure and a function?**

**Basic Difference-**

1. The function must return a value but in**Stored Procedure** it is optional. Even a procedure can return zero or n values.
2. Functions can have only input parameters for it whereas Procedures can have input or output parameters.
3. Functions can be called from Procedure whereas Procedures cannot be called from a Function.

**Advanced Differences-**

* The procedure allows SELECT as well as DML(INSERT/UPDATE/DELETE) statement in it whereas Function allows only SELECT statement in it.
* Procedures cannot be utilized in a SELECT statement whereas Function can be embedded in a SELECT statement.
* Stored Procedures cannot be used in the [SQL](https://en.wikipedia.org/wiki/SQL) statements anywhere in the WHERE/HAVING/SELECT section whereas Function can be.
* Functions that return tables can be treated as another rowset. This can be used in JOINs with other tables.
* Inline Function can be though of as views that take parameters and can be used in JOINs and other Rowset operations.
* An exception can be handled by try-catch block in a Procedure whereas try-catch block cannot be used in a Function.
* We can use Transactions in Procedure whereas we can't use Transactions in Function.

**UNION vs UNION ALL**

The difference between **Union and Union all** is that **Union all** will not eliminate duplicate rows, instead it just pulls **all** the rows from **all** the tables fitting your query specifics **and** combines them into a table. A **UNION** statement effectively does a SELECT DISTINCT on the results set.

**DECODE vs CASE**

**CASE** is a statement while **DECODE** is a function. **CASE** can work with logical operators other than '=' : **DECODE** performs an equality check only. **CASE** is capable of other logical comparisons such as < ,> ,BETWEEN , LIKE etc.

**NVL vs NVL2 vs COALESCE**

**NVL :** This is an Oracle-only function and is probably the most commonly used by Oracle developers. Its purpose is to let you replace NULLs with another value or expression.

This is the syntax: NVL(expression1, expression2)

**NVL2:** This is another Oracle-specific function, which lets you decide which value to return, based on whether a specified expression is null or not. There’s another function that can work in a very similar way (DECODE), but we will talk about it some other day.

This is NVL2’s syntax: NVL2(expression1, expression2, expression3)

**COALESCE :** This function is actually part of the SQL standard, and thus is implemented in most DBMSs. Basically it will return the first non-null value from a list of expressions provided.

This is the syntax: COALESCE(expression-1, expression-2, ... , expression-n)

**Truncate vs Delete**

1. truncate is fast delete is slow.

2. truncate doesn't do logging delete logs on per row basis.

3. rollback is possible with delete not with truncate until specifically supported by the vendor.

4. truncate doesn't fire trigger, delete does.

5. Don't delete, truncate it when it comes to purge tables.

6. truncate reset identity column in table if any, delete doesn't.

7. truncate is DDL while delete is DML (use this when you are writing exam)

8. truncate doesn't support where clause, delete does.

**Clustered vs Non-clustered index**

* Clustered
  + Clustered indexes sort and store the data rows in the table or view based on their key values. These are the columns included in the index definition. There can be only one clustered index per table, because the data rows themselves can be stored in only one order.
  + The only time the data rows in a table are stored in sorted order is when the table contains a clustered index. When a table has a clustered index, the table is called a clustered table. If a table has no clustered index, its data rows are stored in an unordered structure called a heap.
* Nonclustered
  + Nonclustered indexes have a structure separate from the data rows. A nonclustered index contains the nonclustered index key values and each key value entry has a pointer to the data row that contains the key value.
  + The pointer from an index row in a nonclustered index to a data row is called a row locator. The structure of the row locator depends on whether the data pages are stored in a heap or a clustered table. For a heap, a row locator is a pointer to the row. For a clustered table, the row locator is the clustered index key.
  + You can add nonkey columns to the leaf level of the nonclustered index to by-pass existing index key limits, and execute fully covered, indexed, queries. For more information, see [Create Indexes with Included Columns](https://docs.microsoft.com/en-us/sql/relational-databases/indexes/create-indexes-with-included-columns?view=sql-server-ver15). For details about index key limits see [Maximum Capacity Specifications for SQL Server](https://docs.microsoft.com/en-us/sql/sql-server/maximum-capacity-specifications-for-sql-server?view=sql-server-ver15)

**Transaction ACID properties:**

**Atomicity**

All changes to data are performed as if they are a single operation. That is, all the changes are performed, or none of them are.

For example, in an application that transfers funds from one account to another, the atomicity property ensures that, if a debit is made successfully from one account, the corresponding credit is made to the other account.

**Consistency**

Data is in a consistent state when a transaction starts and when it ends.

For example, in an application that transfers funds from one account to another, the consistency property ensures that the total value of funds in both the accounts is the same at the start and end of each transaction.

**Isolation**

The intermediate state of a transaction is invisible to other transactions. As a result, transactions that run concurrently appear to be serialized.

For example, in an application that transfers funds from one account to another, the isolation property ensures that another transaction sees the transferred funds in one account or the other, but not in both, nor in neither.

**Durability**

After a transaction successfully completes, changes to data persist and are not undone, even in the event of a system failure.

For example, in an application that transfers funds from one account to another, the durability property ensures that the changes made to each account will not be reversed.

**ISOLATION levels**

The SQL standard defines four isolation levels :

1. **Read Uncommitted –**Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transaction, thereby allowing dirty reads. In this level, transactions are not isolated from each other.
2. **Read Committed –**This isolation level guarantees that any data read is committed at the moment it is read. Thus it does not allows dirty read. The transaction holds a read or write lock on the current row, and thus prevent other transactions from reading, updating or deleting it.
3. **Repeatable Read –**This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on all rows it inserts, updates, or deletes. Since other transaction cannot read, update or delete these rows, consequently it avoids non-repeatable read.
4. **Serializable –**This is the Highest isolation level. A *serializable* execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

**Materialized Views**

Views are virtual tables composed of the result set of a SQL query and the contents are usually not stored physically.  They allow hiding the complexity of SQL queries thus creating a level of abstraction. You can think over views as a stored query in the server. The main advantage of views is that they can subset the data contained in a table exposing only the interested columns and rows in order to abstract the presentation of the table; they can join and simplify multiple tables into a single virtual table; they can act as aggregated tables; they can restrict direct access to tables by end users, and finally they can provide a standard interface of the data.

Views cannot solve all the problems explained above. For example, if a view performs an aggregation of millions of rows, query performance of the view decreases dramatically and this operation is performed many times every day, the database system must access millions of rows repeatedly. A new data structure must be defined to deal with this kind of scenario. A materialized view is defined just as a regular view but the result set of the query is stored as persistent data object such as table which is frequently updated from the underlying base tables when changes occur. They are useful to aggregate data in business intelligence applications with complex queries.