1.Types of commands and their examples.

Sql commands serve different purposes in managing and manipulating databases. These types of commands include data definition language (DDL), which defines and modifies database structure; data manipulation language (DML), which is used to manipulate data within tables; data control language (DCL), which governs the access privileges and permissions; transactional control language, which manages the transactional aspect of database operations; and data query language (DQL), which allows retrieval and organization of data from a database.
These SQL commands are essential to perform SQL operations and aid developers and database administrators in ensuring data consistency, integrity, and accessibility.

1.1 DDL or Data Definition Language
Data definition language (DDL) is a collection of SQL commands that are used to build, change, and delete database structures. They describe the database schema and are used to design the layout of the objects stored in the database. DDL commands do not affect the data in the database.

Here is the list of DDL commands:

CREATE: It is used in the creation of the database and includes items or objects, such as a table, views, index, or/and stored procedure function, and triggers.
Syntax for Creating a Table:
CREATE TABLE <table_name> (column1 datatype, column2 datatype, ….);
Example:
CREATE TABLE EMPLOYEE(Name VARCHAR2(20), DOB DATE);

DROP: It is used to remove objects from any database. It deletes both the structure and record stored in the table.
Syntax for Dropping a Table:
DROP TABLE <table_name>;
Example:
DROP TABLE EMPLOYEE;

ALTER: This can be utilized to change the database's design or structure. The ALTER command can be used to either modify the characteristics of an existing attribute or add a new attribute altogether.
Syntax to Modify Existing Column in the Table:
ALTER TABLE <table_name> MODIFY(column_definitions....);
Example:
ALTER TABLE EMP_DETAILS MODIFY (NAME VARCHAR2(20));
Syntax to Add a New Column to the Table:
ALTER TABLE table_name ADD column_name COLUMN-definition;
Example:
ALTER TABLE EMP_DETAILS ADD(ADDRESS VARCHAR2(20));

TRUNCATE: This command can be employed to eliminate all records or entries from a table, along with all allotted spaces for those items. Though the data is deleted from the table,

the structure of the table remains. It is often used to clear large amounts of data from a table.
Syntax:
TRUNCATE TABLE <table_name>;
Example:
TRUNCATE TABLE EMPLOYEE

COMMENT: This command is used to insert comments into the data dictionaries. Comments help the developers or administrators to understand the intent and functionality of an object in a database.
Syntax:
COMMENT ON TABLE <table_name> IS 'This is a comment.';
Example:
COMMENT ON TABLE employees IS 'This table stores information about company employees.';

RENAME: It is used while renaming an existing object in the system of the database.
Syntax:
RENAME TABLE <old_table_name> TO <new_table_name>;
Example:
RENAME TABLE sales_data TO monthly_sales;

1.2. DQL or Data Query Language

Data query language (DQL) is a set of SQL commands that are used to query data inside schema objects. It fetches a schema relationship based on the query provided. With these commands, you can retrieve data from the database and create a structure for it.
DQL has a SELECT command and its clauses that allow the user to extract data and execute actions on single or multiple tables. Query results for this command are stored in a temporary table that is later retrieved by the application or the front end to access the content of the database.
Syntax:
SELECT column1, column2, ... FROM <table_name>;
Example:
SELECT first_name, salary FROM employees;
Certain clauses are used together with the SELECT command to retrieve relevant information. These include:

- JOIN Clause: It is used to combine rows from two or more tables based on one related column between them. There are different types of JOIN clauses, including INNER JOIN, RIGHT JOIN, LEFT JOIN, and FULL JOIN.

Syntax: SELECT columns FROM table1 [JOIN TYPE] JOIN table2 ON table1.column_name = table2.column_name;
Example: Here is a SQL query that uses INNER JOIN to retrieve information from the employee table.
SELECT employees.first_name, employees.last_name, departments.department_name
FROM employees
INNER JOIN departments
ON employees.department_id = departments.department_id;

- WHERE Clause: It is used with the SELECT command to filter records based on specific conditions.

Syntax: SELECT column1, column2, … FROM table_name WHERE condition;
Example: SELECT product_name, price FROM products WHERE category = 'Electronics';

- ORDER BY Clause: It is used to sort the results in ascending or descending order.

Syntax: SELECT column1, column2 FROM table_name ORDER BY column1 [ASC|DESC], column2 [ASC|DESC];
Example: Here is an example of an SQL query to select specific columns from the 'students' table.

```
SELECT first_name, last_name
FROM students
ORDER BY age ASC, last_name DESC;
```

- GROUP BY Clause: It is used with aggregate functions to group the result according to one or more columns.

Syntax: SELECT column1, aggregate_function(column2) FROM table_name GROUP BY column1;
Example: Here's an example of an SQL query using the SELECT command with 'SUM' aggregate function to add the sales amounts grouped by 'region'.

```
SELECT region, SUM(amount)
FROM sales
GROUP BY region;
```

1.3. DML or Data Manipulation Language

DML (Data Manipulation Language) refers to SQL instructions that deal with the alteration of information stored within a database, which constitutes the majority of SQL queries. This is a SQL statement component that governs the database and information access. The DCL statements are categorized together with data manipulation statements. Here is the list of DML commands:

Here is the list of DML commands:

INSERT: This command inserts data into an existing table. You can insert multiple rows in a table using this command. You can also insert data either by specifying the column names in the SQL query or by not specifying the column name.
Syntax:

```
INSERT INTO <table_name> (column1, column2, column3, ...) VALUES (value1, value2, value3, ...);
```

Example:

```
INSERT INTO students (first_name, last_name, age) VALUES ('John', 'Doe', 20);
```

UPDATE: This command is employed in updating the existing data in a table. The condition specified in the query decides which row is to be updated. If no condition is specified, all the records are updated. You can update single as well as multiple columns.
Syntax:

```
UPDATE <table_name> SET column1 = value1, column2 = value2, ... WHERE condition;
```

Example: Here's an example of an SQL query to update the salary of an employee:

```
UPDATE employees

SET salary = 60000
```

WHERE employee_id = 101;

DELETE: This command is used to remove records from a database table. WHERE clause is used with this command to specify which rows are to be deleted. If the WHERE clause is not added, all the rows will get deleted.
Syntax:
DELETE FROM <table_name> WHERE condition;
Example: The following SQL query will delete all the rows from the 'employees' table where the 'department_id' is equal to 5.
DELETE FROM employees
WHERE department_id = 5;

LOCK: This command is used for concurrent table control. The LOCK command ensures data consistency and prevents conflicts between transactions.
Syntax:
LOCK TABLE <table_name> [IN lock_mode];
Different lock modes in this command include ROW SHARE, ROW EXCLUSIVE, SHARE, and EXCLUSIVE.
Example:
LOCK TABLE employees IN ROW EXCLUSIVE

CALL: This command invokes a PL/SQL. It can execute a standalone procedure or a procedure defined within a type. The data types of the parameters passed by the CALL statement must be the SQL data types.
Syntax:
CALL procedure_name(argument1, argument2, ...);
Example:
CALL get_employee_info(123);
In the above example, the SQL query calls the stored procedure 'get_employee_info' with the argument '123'. It will retrieve and display the content stored in the procedure with respect to the employee ID 123 from the employees table.

EXPLAIN PLAN: It is used to determine the execution plan that the database follows to execute a specified SQL statement.
Syntax:
EXPLAIN <sql_statement>;
Example:
EXPLAIN SELECT * FROM employees WHERE department_id = 10;
The above example will display the execution plan for the 'SELECT' query.

1.4. DCL or Data Control Language

These SQL commands comprise GRANT and REVOKE, which primarily interact with the database system's rights, permits, and other restrictions. Here is the list of DCL commands:
Here is the list of DCL commands:

GRANT: It grants people database access or special privileges on database objects.

Syntax:
GRANT permission_type ON object_type::object_name TO user_role;
Example:
GRANT SELECT ON TABLE employees TO user_john;

REVOKE: It removes the user's access permissions granted using the GRANT command.
Syntax:
REVOKE permission_type ON object_type::object_name FROM user_role;
Example:
REVOKE SELECT ON TABLE employees FROM user_john;

1.5. TCL or Transaction Control Language

A transaction is a collection of tasks that are executed as a single entity. Each transaction commences with a particular task and finishes when all of the tasks throughout the group are accomplished. The transaction fails when one or more of the tasks misses. As a consequence, a transaction has just two outcomes: either it succeeds or it fails. Here is the list of TCL commands:
Here is the list of TCL commands:

BEGIN: This command initiates a transaction. It marks the beginning of a block of SQL statements that are considered as a single unit.
Syntax:
BEGIN [TRANSACTION];
Example: In the following example, we start the transaction with the 'BEGIN TRANSACTION' command, update the status of specific orders in the 'Orders' table, and then commit the transaction to save the changes.

```
BEGIN TRANSACTION;

-- Update the "Status" column of the "Orders" table

UPDATE Orders

SET Status = 'Shipped'

WHERE OrderID = 123;

COMMIT;
```

COMMIT: This command commits or terminates a transaction. It is used to permanently save changes made during a transaction to the database.
Syntax:
COMMIT;
Example: The following example begins a transaction, performs SQL statements, and commits the changes to the database.

```
BEGIN;

UPDATE employees SET salary = salary * 1.1 WHERE department_id = 10;
```

```
COMMIT;
```

ROLLBACK: This command reverts a transaction if an error appears. It will undo changes made during a transaction and are not saved to the database. .
Syntax:

```
ROLLBACK;
```

Example: In the following code, we will undo the changes with the ROLLBACK command.

```
BEGIN;
UPDATE employees SET salary = salary * 1.1 WHERE department_id = 10;

-- Something went wrong, let's rollback the changes

ROLLBACK;
```

SAVEPOINT: It creates a transaction save point. You can set a point within a transaction to which you can later roll back. It does not rollback the entire transaction but only up to a certain point.
Syntax to Create a Savepoint:

```
SAVEPOINT savepoint_name;
```

Syntax to Rollback to the Savepoint:

```
ROLLBACK TO SAVEPOINT savepoint_name;
```

Example: Refer to the following example to understand how to create and rollback to a savepoint using SQL.

```
BEGIN;

-- Make some updates

UPDATE employees SET salary = salary * 1.1 WHERE department_id = 10;

-- Create a savepoint after the first update

SAVEPOINT update1;

-- Make another set of updates

UPDATE employees SET salary = salary * 1.05 WHERE department_id = 20;

-- Oops, something went wrong, let's roll back to the first update

ROLLBACK TO SAVEPOINT update1;

-- Continue with other operations if needed

-- Commit the transaction

COMMIT;
```

SET TRANSACTION: It defines the transaction's parameters.
Syntax:
SET TRANSACTION [transaction_characteristics];
Example:
SET TRANSACTION ISOLATION LEVEL SERIALISABLE;

2.What is Normalization and denormalisation?

Normalization is used to remove redundant data from the database and to store non-redundant and consistent data into it. It is a process of converting an unnormalised table into a normalised table. Database normalisation is an important process because a poorly designed database table is inconsistent and may create issues while performing operations like insertion, deletion, updating, etc.
The process of Normalization involves resolution of database anomalies, elimination of data redundancy, data dependency, isolation of data, and data consistency. Normalization in databases provides a formal framework to analyse the relations based on the key attributes and their functional dependencies. It reduces the requirements of restructuring of tables.
Denormalisation is used to combine multiple table data into one so that it can be queried quickly. It is a process of storing the join of higher normal form relations in the form of base relation that is in a lower normal form. The primary goal of denormalisation is to achieve the faster execution of the queries.
In the process of denormalisation, the data is integrated into the same database. Denormalisation is mainly used where joins are expensive and queries are executed on the table very frequently. However, there is a drawback of denormalisation, that is, a small wastage of memory.

3.Explain 1NF, 2NF, 3NF.

1NF: Every column in the table contains only atomic values (meaning they cannot be divided). Each column has only one value for each row in the table.
2NF: The database satisfies the conditions of 1NF, plus all non-key attributes are functionally dependent on the primary key.
3NF: Relations are in 3NF if they are in 2NF and have no transitive dependencies (meaning no non-key columns are dependent on other non-key columns).

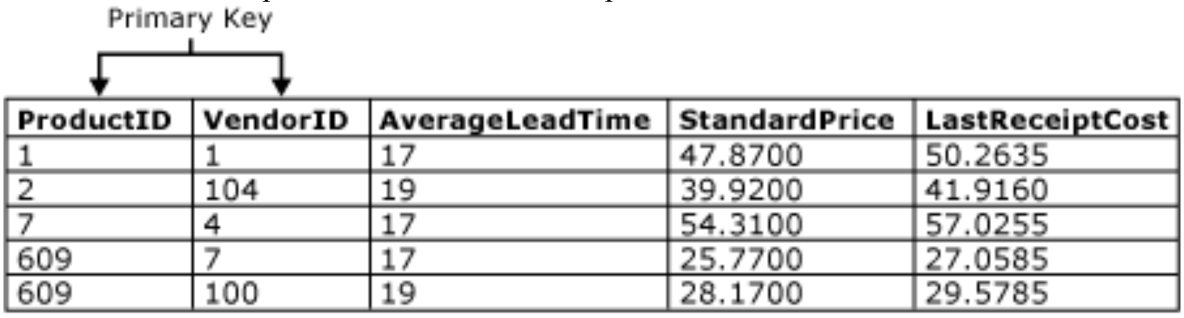
4.Share use case where you had to do denormalization in database..

Denormalization is when developers intentionally add redundancy to normalized relational databases. The purpose of denormalization is to improve the database's read performance in cases in which query speed is more important than a consistent data structure or space optimization.
Some actions developers might take to conduct denormalization include:

- Using extra attributes in a table
- Adding a new table
- Creating instances of tables

5.What is primary key and foreign key?

A table typically has a column or combination of columns that contain values that uniquely identify each row in the table. This column, or columns, is called the primary key (PK) of the table and enforces the entity integrity of the table. Because primary key constraints guarantee unique data, they're frequently defined on an identity column.
When you specify a primary key constraint for a table, the Database Engine enforces data uniqueness by automatically creating a unique index for the primary key columns. This index also permits fast access to data when the primary key is used in queries. If a primary key constraint is defined on more than one column, values may be duplicated within one column, but each combination of values from all the columns in the primary key constraint definition must be unique.
As shown in the following illustration, the ProductID and VendorID columns in the Purchasing.ProductVendor table form a composite primary key constraint for this table. This makes sure that every row in the ProductVendor table has a unique combination of ProductID and VendorID. This prevents the insertion of duplicate rows.


Primary Key

ProductID	VendorID	AverageLeadTime	StandardPrice	LastReceiptCost
1	1	17	47.8700	50.2635
2	104	19	39.9200	41.9160
7	4	17	54.3100	57.0255
609	7	17	25.7700	27.0585
609	100	19	28.1700	29.5785

ProductVendor table

A table can contain only one primary key constraint.

A primary key can't exceed 16 columns and a total key length of 900 bytes.

The index generated by a primary key constraint can't cause the number of indexes on the table to exceed 999 nonclustered indexes and 1 clustered index.

If clustered or nonclustered isn't specified for a primary key constraint, clustered is used if there's no clustered index on the table.

All columns defined within a primary key constraint must be defined as not null. If nullability isn't specified, all columns participating in a primary key constraint have their nullability set to not null.

If a primary key is defined on a CLR user-defined type column, the implementation of the type must support binary ordering.

Foreign Key Constraints
A foreign key (FK) is a column or combination of columns that is used to establish and enforce a link between the data in two tables to control the data that can be stored in the foreign key table. In a foreign key reference, a link is created between two tables when the column or columns that hold the primary key value for one table are referenced by the column or columns in another table. This column becomes a foreign key in the second table.
For example, the Sales.SalesOrderHeader table has a foreign key link to the Sales.SalesPerson table because there's a logical relationship between sales orders and salespeople. The SalesPersonID column in the SalesOrderHeader table matches the primary

key column of the SalesPerson table. The SalesPersonID column in the SalesOrderHeader table is the foreign key to the SalesPerson table. By creating this foreign key relationship, a value for SalesPersonID can't be inserted into the SalesOrderHeader table if it doesn't already exist in the SalesPerson table.
A table can reference a maximum of 253 other tables and columns as foreign keys (outgoing references). SQL Server 2016 (13.x) increases the limit for the number of other tables and columns that can reference columns in a single table (incoming references), from 253 to 10,000. (Requires at least 130 compatibility level.) The increase has the following restrictions:

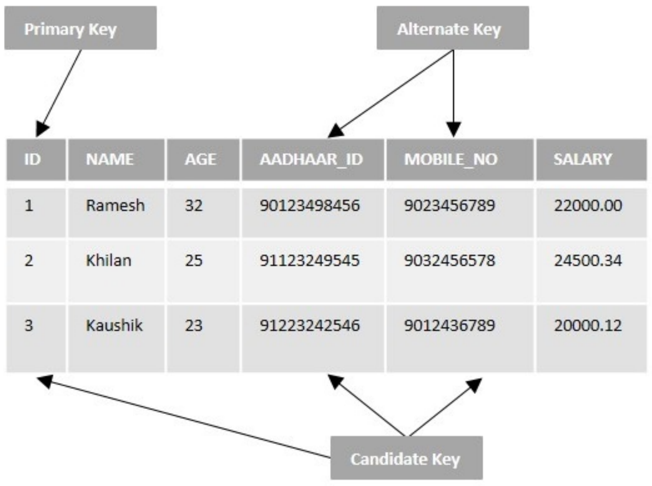
Greater than 253 foreign key references are only supported for DELETE DML operations. UPDATE and MERGE operations aren't supported.

A table with a foreign key reference to itself is still limited to 253 foreign key references.

Greater than 253 foreign key references aren't currently available for columnstore indexes, memory-optimized tables, Stretch Database, or partitioned foreign key tables.

6.what is alternate and candidate key?

SQL Alternate Keys in a database table are candidate keys that are not currently selected as a primary key. They can be used to uniquely identify a tuple(or a record) in a table.
There is no specific query or syntax to set the alternate key in a table. It is just a column that is a close second candidate which could be selected as a primary key. Hence, they are also called secondary candidate keys.
If a database table consists of only one candidate key, that is treated as the primary key of the table, then there is no alternate key in that table.
Let us understand the concept of alternate key with an example. Suppose we have a table named CUSTOMERS with various fields like ID, NAME, AGE, AADHAAR_ID, MOBILE_NO and SALARY as shown below.



ID	NAME	AGE	AADHAAR_ID	MOBILE_NO	SALARY
1	Ramesh	32	90123498456	9023456789	22000.00
2	Khilan	25	91123249545	9032456578	24500.34
3	Kaushik	23	91223242546	9012436789	20000.12

The details like id, mobile number and aadhaar number of a customer are unique, and we can identify the records from the CUSTOMERS table uniquely using their respective fields; ID, AADHAAR_ID and MOBILE_NO. Therefore, these three fields can be treated as candidate keys.
And among them, if one is declared as the primary key of the CUSTOMERS table then the remaining two would be alternate keys.

Features of Alternate Keys
Following are some important properties/features of alternate keys −

The alternate key does not allow duplicate values.
A table can have more than one alternate keys.
The alternate key can contain NULL values unless the NOT NULL constraint is set explicitly.
All alternate keys can be candidate keys, but all candidate keys can not be alternate keys.
The primary key, which is also a candidate key, can not be considered as an alternate key.

Candidate Key
A Candidate key is a subset of super keys that is used to uniquely identify records of a table. It can be a single field or multiple fields. The primary keys, alternate keys, foreign keys in a table are all types of candidate key.

7.What are window functions?

MySQL also supports nonaggregate functions that are used only as window functions. For these, the OVER clause is mandatory:
CUME_DIST()
DENSE_RANK()
FIRST_VALUE()
LAG()
LAST_VALUE()
LEAD()
NTH_VALUE()
NTILE()
PERCENT_RANK()
RANK()
ROW_NUMBER()

Name	Description
CUME_DIST()	Cumulative distribution value
DENSE_RANK()	Rank of current row within its partition, without gaps
FIRST_VALUE()	Value of argument from first row of window frame
LAG()	Value of argument from row lagging current row within partition
LAST_VALUE()	Value of argument from last row of window frame
LEAD()	Value of argument from row leading current row within partition
NTH_VALUE()	Value of argument from N-th row of window frame
NTILE()	Bucket number of current row within its partition.
PERCENT_RANK()	Percentage rank value

RANK()	Rank of current row within its partition, with gaps
ROW_NUMBER()	Number of current row within its partition

8.Explain Ranking Functions? GIven a small table , write the output.

We perform calculations on data using various aggregated functions such as Max, Min, and AVG. We get a single output row using these functions. SQL provides SQL RANK functions to specify rank for individual fields as per the categorisations. It returns an aggregated value for each participating row. SQL RANK functions also knows as Window Functions.

We have the following rank functions.

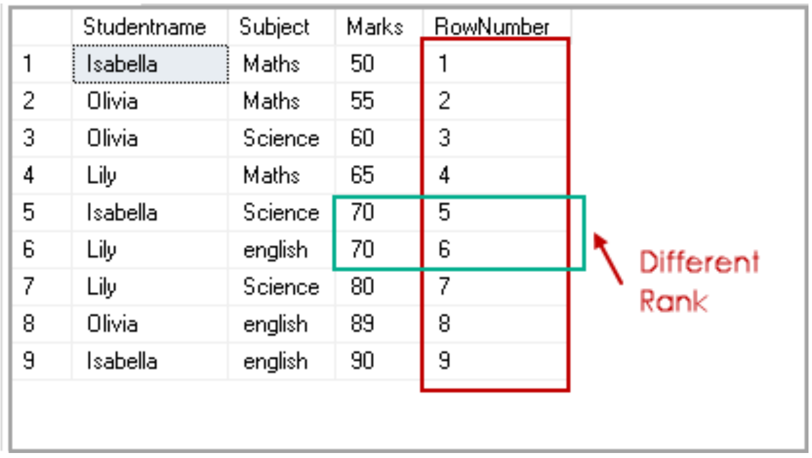
- ROW_NUMBER()
- RANK()
- DENSE_RANK()
- NTILE()

In the SQL RANK functions, we use the OVER() clause to define a set of rows in the result set. We can also use SQL PARTITION BY clause to define a subset of data in a partition. You can also use Order by clause to sort the results in a descending or ascending order.

ROW_Number() SQL RANK function

We use ROW_Number() SQL RANK function to get a unique sequential number for each row in the specified data. It gives the rank one for the first row and then increments the value by one for each row. We get different ranks for the row having similar values as well.

Studentname,	Subject,	Marks,	RowNumber
Isabella,	Maths,	50,	1
Olivia,	Maths,	55,	2
Olivia,	Science,	60,	3
Lily,	Maths,	65,	4
Lily,	english,	70,	5
Isabella,	Science,	70,	6
Lily,	Science,	80,	7
Olivia,	english,	89,	8
Isabella,	english,	90,	9

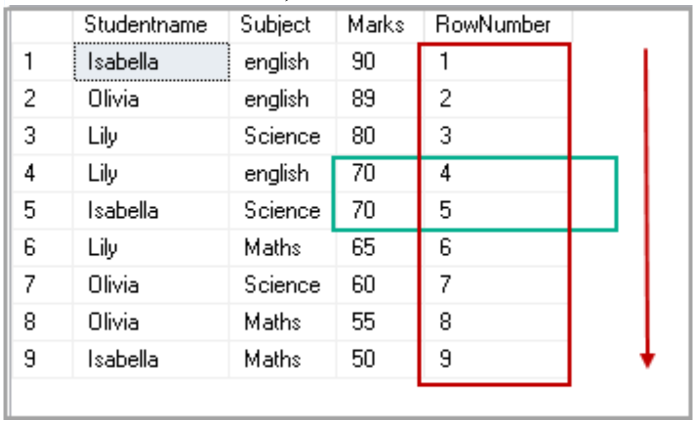


	Studentname	Subject	Marks	RowNumber
1	Isabella	Maths	50	1
2	Olivia	Maths	55	2
3	Olivia	Science	60	3
4	Lily	Maths	65	4
5	Isabella	Science	70	5
6	Lily	english	70	6
7	Lily	Science	80	7
8	Olivia	english	89	8
9	Isabella	english	90	9

By default, it sorts the data in ascending order and starts assigning ranks for each row. In the

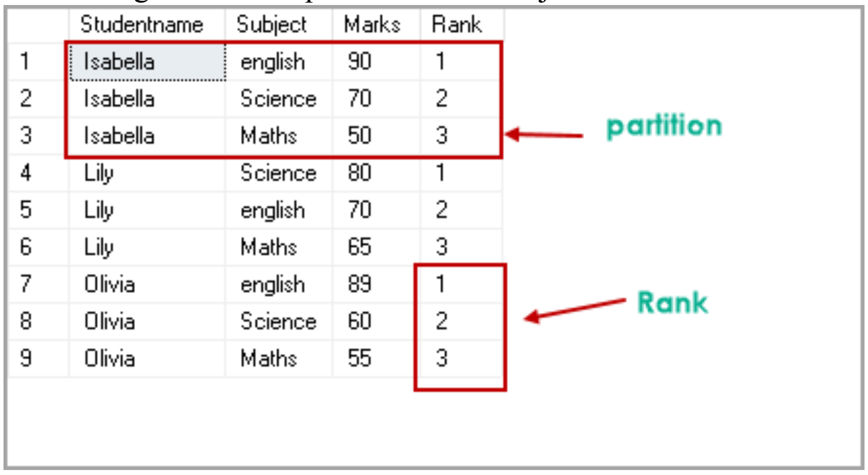
above screenshot, we get ROW number 1 for marks 50.
We can specify descending order with Order By clause, and it changes the RANK accordingly.

```
SELECT Studentname,
       Subject,
       Marks,
       ROW_NUMBER() OVER(ORDER BY Marks desc) RowNumber
FROM ExamResult;
```



	Studentname	Subject	Marks	RowNumber
1	Isabella	english	90	1
2	Olivia	english	89	2
3	Lily	Science	80	3
4	Lily	english	70	4
5	Isabella	Science	70	5
6	Lily	Maths	65	6
7	Olivia	Science	60	7
8	Olivia	Maths	55	8
9	Isabella	Maths	50	9

RANK() SQL RANK Function
We use RANK() SQL Rank function to specify rank for each row in the result set. We have student results for three subjects. We want to rank the result of students as per their marks in the subjects. For example, in the following screenshot, student Isabella got the highest marks in English subject and lowest marks in Maths subject. As per the marks, Isabella gets the first rank in English and 3rd place in Maths subject.



	Studentname	Subject	Marks	Rank
1	Isabella	english	90	1
2	Isabella	Science	70	2
3	Isabella	Maths	50	3
4	Lily	Science	80	1
5	Lily	english	70	2
6	Lily	Maths	65	3
7	Olivia	english	89	1
8	Olivia	Science	60	2
9	Olivia	Maths	55	3

Execute the following query to get this result set. In this query, you can note the following things:

- We use PARTITION BY Studentname clause to perform calculations on each student group
- Each subset should get rank as per their Marks in descending order
- The result set uses Order By clause to sort results on Studentname and their rank

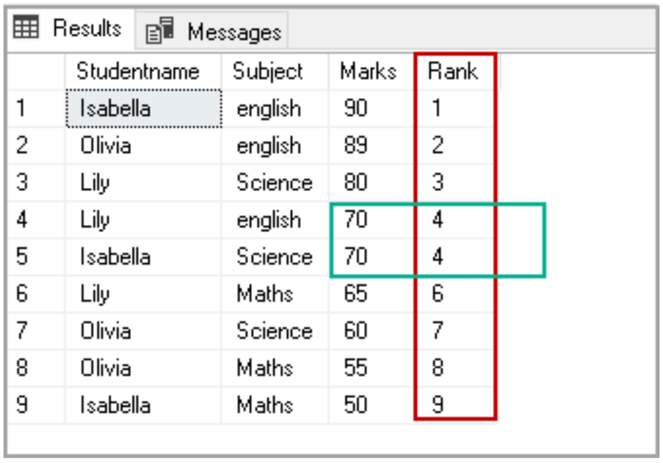
```
SELECT Studentname,
       Subject,
       Marks,
       RANK() OVER(PARTITION BY Studentname ORDER BY Marks DESC) Rank
FROM ExamResult
ORDER BY Studentname,
         Rank;
```

Let’s execute the following query of SQL Rank function and look at the result set. In this query, we did not specify SQL PARTITION By clause to divide the data into a smaller subset. We use SQL Rank function with over clause on Marks clause (in descending order) to get ranks for respective rows.

```
SELECT Studentname,
       Subject,
       Marks,
       RANK() OVER(ORDER BY Marks DESC) Rank
FROM ExamResult
ORDER BY Rank;
```

In the output, we can see each student get rank as per their marks irrespective of the specific subject. For example, the highest and lowest marks in the complete result set are 90 and 50 respectively. In the result set, the highest mark gets RANK 1, and the lowest mark gets RANK 9.

If two students get the same marks (in our example, ROW numbers 4 and 5), their ranks are also the same.



	Studentname	Subject	Marks	Rank
1	Isabella	english	90	1
2	Olivia	english	89	2
3	Lily	Science	80	3
4	Lily	english	70	4
5	Isabella	Science	70	4
6	Lily	Maths	65	6
7	Olivia	Science	60	7
8	Olivia	Maths	55	8
9	Isabella	Maths	50	9

DENSE_RANK() SQL RANK function

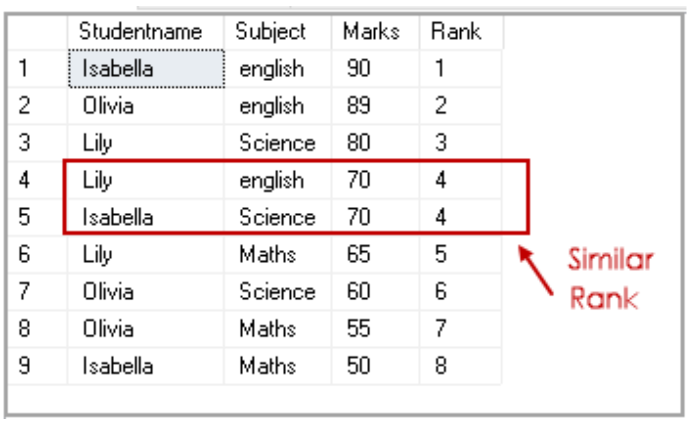
We use DENSE_RANK() function to specify a unique rank number within the partition as per the specified column value. It is similar to the Rank function with a small difference.

In the SQL RANK function DENSE_RANK(), if we have duplicate values, SQL assigns different ranks to those rows as well. Ideally, we should get the same rank for duplicate or similar values.

Let’s execute the following query with the DENSE_RANK() function.

```
SELECT Studentname,
       Subject,
       Marks,
       DENSE_RANK() OVER(ORDER BY Marks DESC) Rank
FROM ExamResult
ORDER BY Rank;
```

In the output, you can see we have the same rank for both Lily and Isabella who scored 70 marks.

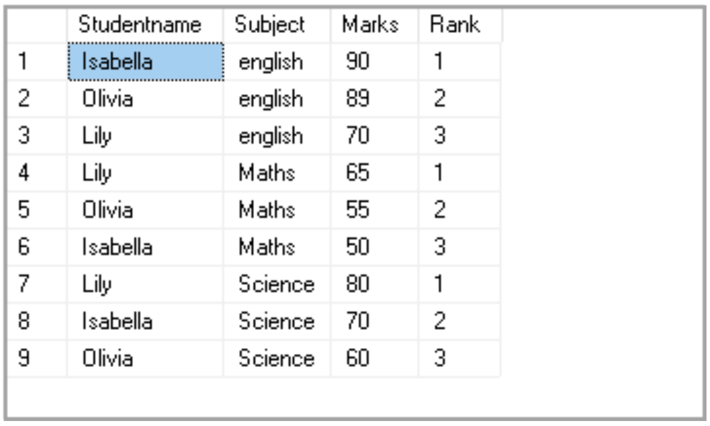


	Studentname	Subject	Marks	Rank
1	Isabella	english	90	1
2	Olivia	english	89	2
3	Lily	Science	80	3
4	Lily	english	70	4
5	Isabella	Science	70	4
6	Lily	Maths	65	5
7	Olivia	Science	60	6
8	Olivia	Maths	55	7
9	Isabella	Maths	50	8

Let’s use DENSE_RANK function in combination with the SQL PARTITION BY clause.

```
SELECT Studentname,
       Subject,
       Marks,
       DENSE_RANK() OVER(PARTITION BY Subject ORDER BY Marks DESC) Rank
FROM ExamResult
ORDER BY Studentname,
         Rank;
```

We do not have two students with similar marks; therefore result set similar to RANK Function in this case.

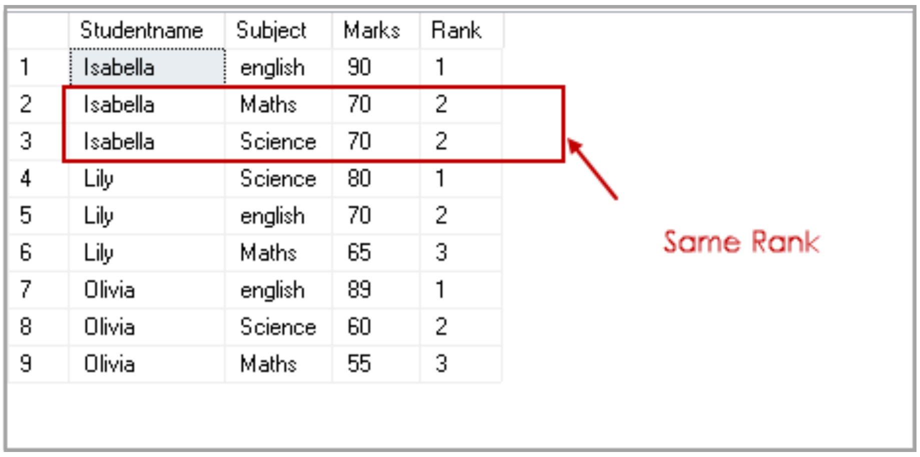


	Studentname	Subject	Marks	Rank
1	Isabella	english	90	1
2	Olivia	english	89	2
3	Lily	english	70	3
4	Lily	Maths	65	1
5	Olivia	Maths	55	2
6	Isabella	Maths	50	3
7	Lily	Science	80	1
8	Isabella	Science	70	2
9	Olivia	Science	60	3

Let’s update the student mark with the following query and rerun the query.

```
Update Examresult set Marks=70 where Studentname='Isabella' and Subject='Maths'
```

We can see that in the student group, Isabella got similar marks in Maths and Science subjects. Rank is also the same for both subjects in this case.



	Studentname	Subject	Marks	Rank
1	Isabella	english	90	1
2	Isabella	Maths	70	2
3	Isabella	Science	70	2
4	Lily	Science	80	1
5	Lily	english	70	2
6	Lily	Maths	65	3
7	Olivia	english	89	1
8	Olivia	Science	60	2
9	Olivia	Maths	55	3

Let’s see the difference between RANK() and DENSE_RANK() SQL Rank function with the following query.

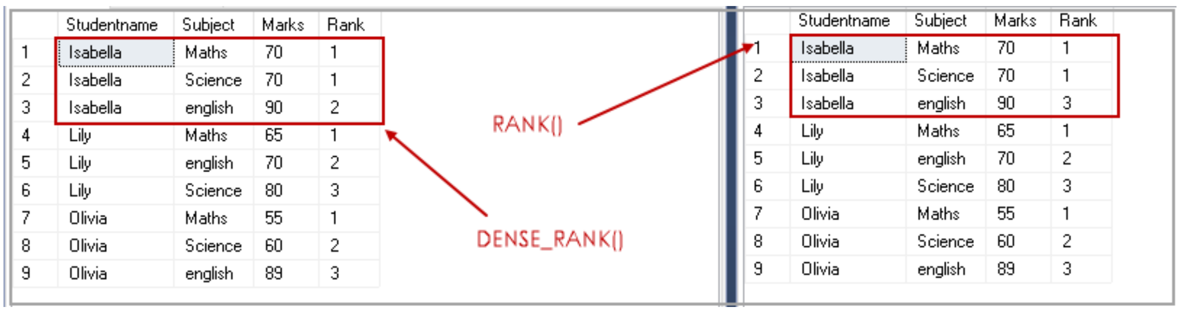
- Query 1

```
SELECT Studentname,
       Subject,
       Marks,
       RANK() OVER(PARTITION BY StudentName ORDER BY Marks ) Rank
FROM ExamResult
ORDER BY Studentname,
         Rank;
```

- Query 2

```
SELECT Studentname,
       Subject,
       Marks,
       DENSE_RANK() OVER(PARTITION BY StudentName ORDER BY
Marks ) Rank
FROM ExamResult
ORDER BY Studentname,
         Rank;
```

In the output, you can see a gap in the rank function output within a partition. We do not have any gap in the DENSE_RANK function.

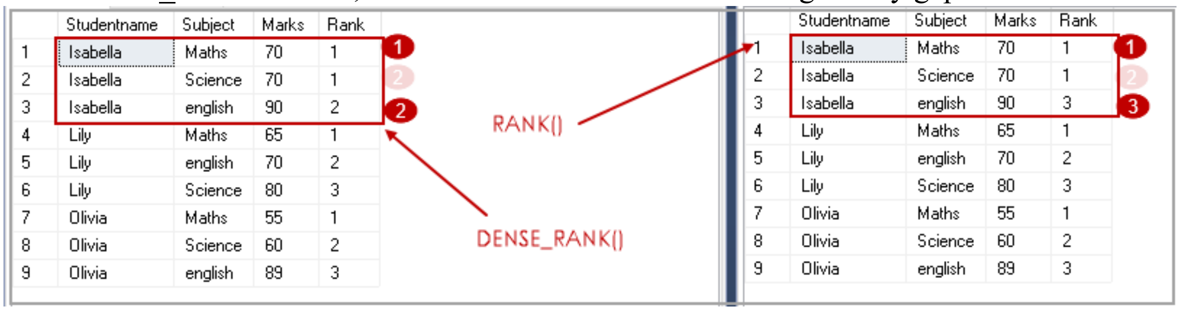


	Studentname	Subject	Marks	Rank
1	Isabella	Maths	70	1
2	Isabella	Science	70	1
3	Isabella	english	90	2
4	Lily	Maths	65	1
5	Lily	english	70	2
6	Lily	Science	80	3
7	Olivia	Maths	55	1
8	Olivia	Science	60	2
9	Olivia	english	89	3

	Studentname	Subject	Marks	Rank
1	Isabella	Maths	70	1
2	Isabella	Science	70	1
3	Isabella	english	90	3
4	Lily	Maths	65	1
5	Lily	english	70	2
6	Lily	Science	80	3
7	Olivia	Maths	55	1
8	Olivia	Science	60	2
9	Olivia	english	89	3

In the following screenshot, you can see that Isabella has similar numbers in the two subjects. A rank function assigns rank 1 for similar values however, internally ignores rank two, and the next row gets rank three.

In the Dense_Rank function, it maintains the rank and does not give any gap for the values.



	Studentname	Subject	Marks	Rank
1	Isabella	Maths	70	1
2	Isabella	Science	70	1
3	Isabella	english	90	2
4	Lily	Maths	65	1
5	Lily	english	70	2
6	Lily	Science	80	3
7	Olivia	Maths	55	1
8	Olivia	Science	60	2
9	Olivia	english	89	3

	Studentname	Subject	Marks	Rank
1	Isabella	Maths	70	1
2	Isabella	Science	70	1
3	Isabella	english	90	3
4	Lily	Maths	65	1
5	Lily	english	70	2
6	Lily	Science	80	3
7	Olivia	Maths	55	1
8	Olivia	Science	60	2
9	Olivia	english	89	3

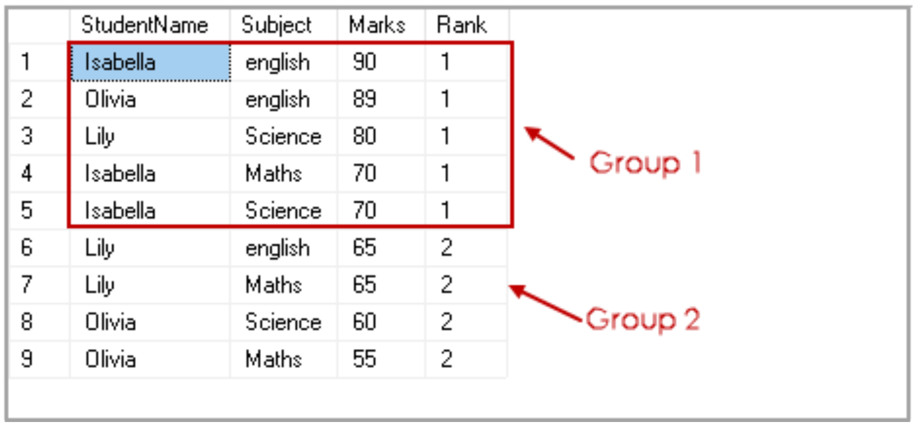
NTILE(N) SQL RANK function

We use the NTILE(N) function to distribute the number of rows in the specified (N) number of groups. Each row group gets its rank as per the specified condition. We need to specify the value for the desired number of groups.

In my example, we have nine records in the ExamResult table. The NTILE(2) shows that we require a group of two records in the result.

```
SELECT *,
       NTILE(2) OVER(
       ORDER BY Marks DESC) Rank
FROM ExamResult
ORDER BY rank;
```

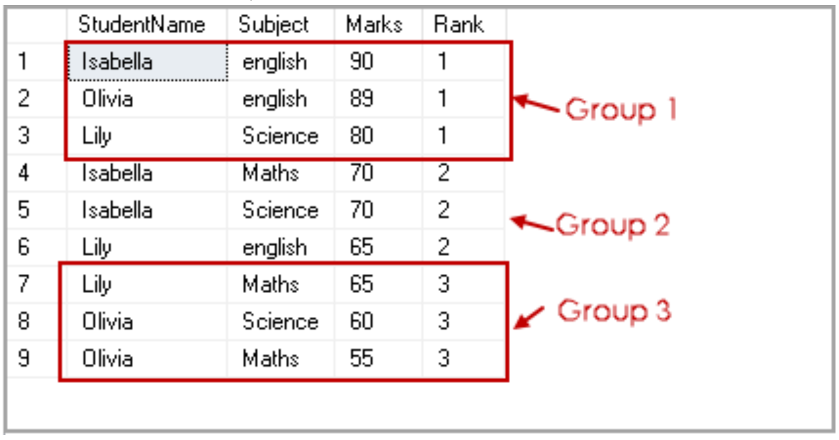
In the output, we can see two groups. Group 1 contains five rows, and Group 2 contains four rows.



	StudentName	Subject	Marks	Rank
1	Isabella	english	90	1
2	Olivia	english	89	1
3	Lily	Science	80	1
4	Isabella	Maths	70	1
5	Isabella	Science	70	1
6	Lily	english	65	2
7	Lily	Maths	65	2
8	Olivia	Science	60	2
9	Olivia	Maths	55	2

Similarly, NTILE(3) divides the number of rows of three groups having three records in each group.

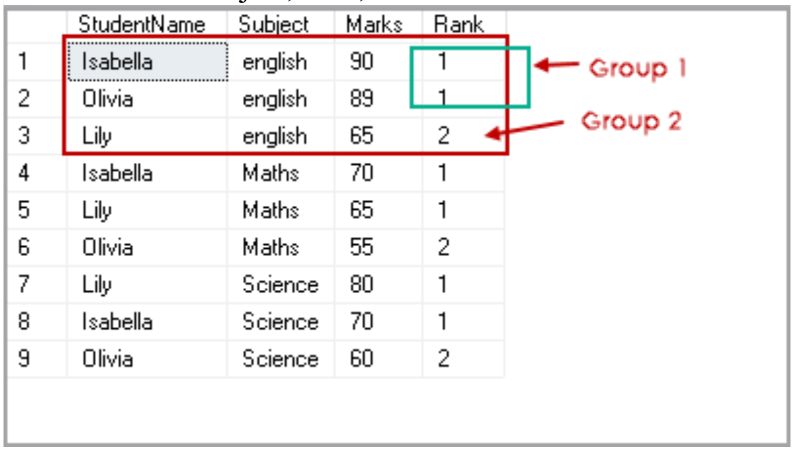
```
SELECT *,
       NTILE(3) OVER(
       ORDER BY Marks DESC) Rank
FROM ExamResult
ORDER BY rank;
```



	StudentName	Subject	Marks	Rank
1	Isabella	english	90	1
2	Olivia	english	89	1
3	Lily	Science	80	1
4	Isabella	Maths	70	2
5	Isabella	Science	70	2
6	Lily	english	65	2
7	Lily	Maths	65	3
8	Olivia	Science	60	3
9	Olivia	Maths	55	3

We can use SQL PARTITION BY clause to have more than one partition. In the following query, each partition on subjects is divided into two groups.

```
SELECT *,
       NTILE(2) OVER(PARTITION  BY subject ORDER BY Marks DESC) Rank
FROM ExamResult
ORDER BY subject, rank;
```



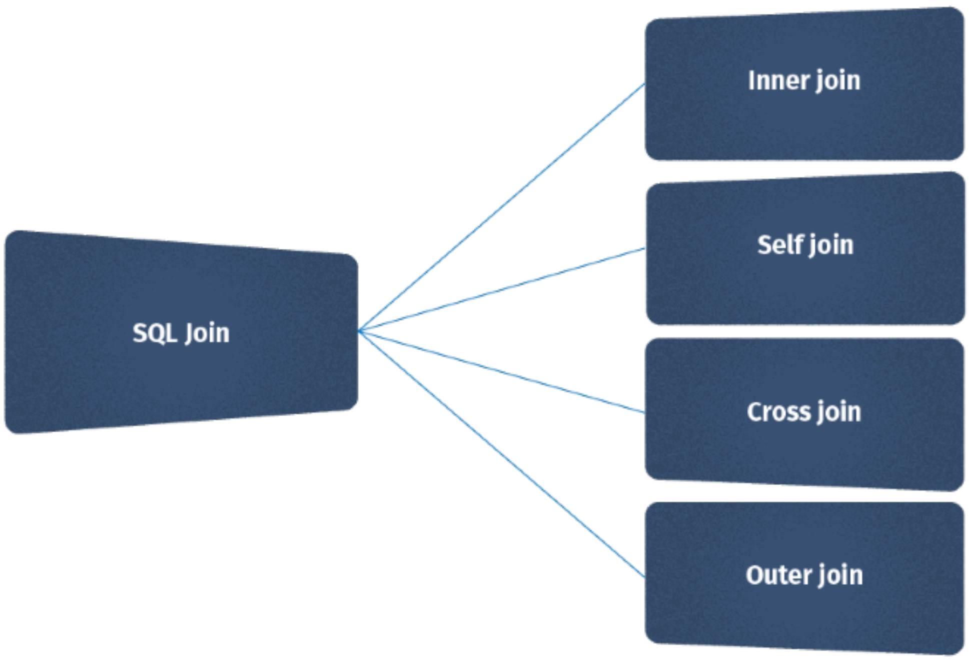
	StudentName	Subject	Marks	Rank
1	Isabella	english	90	1
2	Olivia	english	89	1
3	Lily	english	65	2
4	Isabella	Maths	70	1
5	Lily	Maths	65	1
6	Olivia	Maths	55	2
7	Lily	Science	80	1
8	Isabella	Science	70	1
9	Olivia	Science	60	2

9.Types of Joins? With example and usecase. All the number of records return and exact records.

SQL Server supports many kinds of different joins including INNER JOIN, SELF JOIN, CROSS JOIN, and OUTER JOIN. In fact, each join type defines the way two tables are related in a query. OUTER JOINS can further be divided into LEFT OUTER JOINS, RIGHT OUTER JOINS, and FULL OUTER JOINS.

SQL INNER JOIN creates a result table by combining rows that have matching values in two or more tables.
SQL LEFT OUTER JOIN includes in a result table unmatched rows from the table that is specified before the LEFT OUTER JOIN clause.
SQL RIGHT OUTER JOIN creates a result table and includes into it all the records

from the right table and only matching rows from the left table.
SQL SELF JOIN joins the table to itself and allows comparing rows within the same table.
SQL CROSS JOIN creates a result table containing paired combination of each row of the first table with each row of the second table.



Inner Join

matching values and is used to retrieve data that appears in both tables.

```
SELECT t1.table_id, t1.name,t2.color
FROM tableA t1
INNER JOIN tableB t2
ON t1.table_id = t2.tableB_id;
```

```
table_id;name;color
1;apple;Red
1;apple;Red
1;apple;Red
1;apple;Red
2;Banana;yellow
2;Banana;yellow
3;Guava;Green
```

Outer Join
Outer Join is used to join multiple database tables into a combined result-set, that includes all the records, even if they don't satisfy the join condition. NULL values are displayed against these records where the join condition is not met.

This scenario only occurs if the left table (or the first table) has more records than the right table (or the second table), or vice versa.

There are three types of outer joins, namely −

- Left (Outer) Join: Retrieves all the records from the first table, Matching records from the second table and NULL values in the unmatched rows.
- Right (Outer) Join: Retrieves all the records from the second table, Matching records from the first table and NULL values in the unmatched rows.
- Full (Outer) Join: Retrieves records from both the tables and fills the unmatched values with NULL.

The SQL Left Join

Left Join or Left Outer Join in SQL combines two or more tables, where the first table is returned wholly; but, only the matching record(s) are retrieved from the consequent tables. If zero (0) records are matched in the consequent tables, the join will still return a row in the result, but with NULL in each column from the right table.

```
SELECT t1.table_id, t1.name,t2.color
FROM tableA t1
LEFT JOIN tableB t2
ON t1.table_id = t2.tableB_id;
```

```
table_id;name;color
1;apple;Red
1;apple;Red
1;apple;Red
1;apple;Red
2;Banana;yellow
2;Banana;yellow
3;Guava;Green
```

The SQL Right Join

The Right Join or Right Outer Join query in SQL returns all rows from the right table, even if there are no matches in the left table. In short, a right join returns all the values from the right table, plus matched values from the left table or NULL in case of no matching join predicate.

```
SELECT t1.table_id, t1.name,t2.color
FROM tableA t1
RIGHT JOIN tableB t2
ON t1.table_id = t2.tableB_id;
```

```
table_id;name;color
1;apple;Red
1;apple;Red
1;apple;Red
1;apple;Red
2;Banana;yellow
2;Banana;yellow
```

```
3;Guava;Green
```

The SQL Cross Join

An SQL Cross Join is a basic type of inner join that is used to retrieve the Cartesian product (or cross product) of two individual tables. That means, this join will combine each row of the first table with each row of second table (i.e. permutations).

```
SELECT t1.table_id, t1.name,t2.color
FROM tableA t1
CROSS JOIN tableB t2;
table_id,name,color
1,apple,Green
1,apple,yellow
1,apple,Red
1,apple,Green
1,apple,yellow
1,apple,Red
1,apple,Green
1,apple,yellow
1,apple,Red
1,apple,Green
1,apple,yellow
1,apple,Red
2,Banana,Green
2,Banana,yellow
2,Banana,Red
2,Banana,Green
2,Banana,yellow
2,Banana,Red
3,Guava,Green
3,Guava,yellow
3,Guava,Red
```

SQL SELF JOIN

A self join is a regular join, but the table is joined with itself.

```
SELECT t1.table_id, t1.name,t2.color
FROM tableA t1, tableB t2
WHERE t1.table_id = t2.tableB_id;
table_id;name;color
1;apple;Red
1;apple;Red
1;apple;Red
1;apple;Red
2;Banana;yellow
2;Banana;yellow
3;Guava;Green
```

10.Use case when self join is required.

A SQL SELF JOIN is a type of join operation where a table is joined with itself. It allows you to combine data from a single table by creating a virtual copy of the table and establishing relationships between the original and virtual tables. Self joins are used to compare or combine data within the same table, often by creating relationships between rows within the table.
You would use a SQL SELF JOIN when you want to perform comparisons or retrieve data from a single table that has some form of hierarchy, relationships, or dependencies. Common use cases for SELF JOIN include:

Hierarchical data: When dealing with hierarchical structures, such as organizational charts or category trees, to retrieve parent-child relationships.

Versioning or tracking changes: For tracking changes in data, like version history or order tracking, by comparing records in the same table.

Recursive queries: To perform recursive operations, like finding all descendants or ancestors of a particular record.

Network or graph data: For working with data where nodes or entities have connections to other nodes in the same table.

The syntax for a SQL SELF JOIN operation is as follows:

```
SELECT t1.column, t2.column
FROM table AS t1
JOIN table AS t2 ON t1.related_column = t2.related_column;
```

t1 and t2: Aliases for the same table to differentiate between the original and virtual copies.

related_column: The column(s) used to establish relationships within the same table.

```
SELECT e.employee_id, e.employee_name, m.employee_name AS manager_name
FROM employees AS e
LEFT JOIN employees AS m ON e.manager_id = m.employee_id;
```

11.What is subquery?
Subqueries (also known as inner queries or nested queries) are a tool for performing operations in multiple steps. For example, if you wanted to take the sums of several columns, then average all of those values, you'd need to do each aggregation in a distinct step.
Subqueries can be used in several places within a query, but it's easiest to start with the FROM statement. Here's an example of a basic subquery:

```
SELECT sub.*
  FROM (
      SELECT *
        FROM tutorial.sf_crime_incidents_2014_01
       WHERE day_of_week = 'Friday'
      ) sub
```

```
 WHERE sub.resolution = 'NONE'
```

Let's break down what happens when you run the above query:

First, the database runs the "inner query"—the part between the parentheses:

```
SELECT *
  FROM tutorial.sf_crime_incidents_2014_01
 WHERE day_of_week = 'Friday'
```

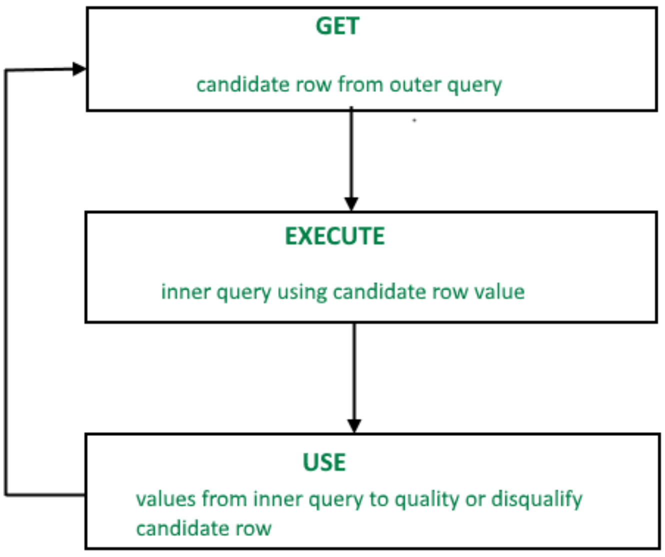
If you were to run this on its own, it would produce a result set like any other query. It might sound like a no-brainer, but it's important: your inner query must actually run on its own, as the database will treat it as an independent query. Once the inner query runs, the outer query will run using the results from the inner query as its underlying table:

```
SELECT sub.*
  FROM (
       <<results from inner query go here>>
       ) sub
 WHERE sub.resolution = 'NONE'
```

Subqueries are required to have names, which are added after parentheses the same way you would add an alias to a normal table. In this case, we've used the name "sub."

12.What is corelated subquery?

Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.


GET
candidate row from outer query
EXECUTE
inner query using candidate row value
USE
values from inner query to quality or disqualify candidate row

A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a SELECT, UPDATE, or DELETE statement.

Syntax

```
SELECT column1, column2, ....
FROM table1 outer
WHERE column1 operator
                (SELECT column1, column2
                 FROM table2
                 WHERE expr1 =
                         outer.expr2);
```

A correlated subquery is one way of reading every row in a table and comparing values in each row against related data. It is used whenever a subquery must return a different result or set of results for each candidate row considered by the main query. In other words, you can use a correlated subquery to answer a multipart question whose answer depends on the value in each row processed by the parent statement.

13.What is CTE?

The common table expression (CTE) is a powerful construct in SQL that helps simplify a query. CTEs work as virtual tables (with records and columns), created during the execution of a query, used by the query, and eliminated after query execution. CTEs often act as a bridge to transform the data in source tables to the format expected by the query.
A common table expression, or CTE, is a temporary named result set created from a simple SELECT statement that can be used in a subsequent SELECT statement. Each SQL CTE is like a named query, whose result is stored in a virtual table (a CTE) to be referenced later in the main query.
The best way to learn common table expressions is through practice. I recommend LearnSQL.com's interactive Recursive Queries course. It contains over 100 exercises that teach CTEs starting with the basics and progressing to advanced topics like recursive common table expressions.

```
WITH my_cte AS (
  SELECT a,b,c
  FROM T1
)
SELECT a,c
FROM my_cte
WHERE ....
```

14.What is derived table?
A derived table is a technique for creating a temporary set of records which can be used within another query in SQL. You can use derived tables to shorten long queries, or even just to break a complex process into logical steps.
A derived table is an example of a subquery that is used in the FROM clause of a SELECT statement to retrieve a set of records. You can use derived tables to break a complex query into separate logical steps and they are often a neat alternative to using temporary tables.

15.Find third highest employee based on salary?

```
        SELECT * FROM `employee` ORDER BY `salary` DESC LIMIT 1 OFFSET 2;
```

16.Find third highest employee based on salary per department?

SELECT * FROM `employee` WHERE GROUP BY ‘dept’ ORDER BY `salary` DESC LIMIT 1 OFFSET 2;

17.How to find duplicate values in a single column?

SELECT dept FROM `employee` GROUP by dept HAVING COUNT(dept)< 1

18.How to find duplicate values in a multiple column?

SELECT dept, emp_name FROM `employee` GROUP by dept,emp_name HAVING COUNT(dept)< 1 AND COUNT(emp_name) < 1;

19.What are ACID properties? give example for each property

A tranaction is a single logical unit of work that accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.
In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called ACID properties.
Atomicity:
By this, we mean that either the entire transaction takes place at once or doesn’t happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.
—Abort: If a transaction aborts, changes made to the database are not visible.
—Commit: If a transaction commits, changes made are visible.
Atomicity is also known as the ‘All or nothing rule’.
Consistency:
This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above,
The total amount before and after the transaction must be maintained.
Isolation:
This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of the database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.
Durability:
This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.

20.Diff between union and union all
UNION and UNION ALL are SQL operators used to concatenate 2 or more result sets. This allows us to write multiple SELECT statements, retrieve the desired results, then combine

them together into a final, unified set.
The main difference between UNION and UNION ALL is that:

UNION: only keeps unique records
UNION ALL: keeps all records, including duplicates

21.Diff between primary key and unique key

The main difference between a primary key vs unique key is that a primary key is a key that uniquely identifies each record in a table but cannot store NULL values. In contrast, a unique key prevents duplicate values in a column and can store NULL values.

22.Diff between truncate and delete

1. The DELETE statement is used when we want to remove some or all of the records from the table, while the TRUNCATE statement will delete entire rows from a table.
2. DELETE is a DML command as it only modifies the table data, whereas the TRUNCATE is a DDL command.
3. DELETE command can filter the record/tuples by using the WHERE clause. However, the TRUNCATE command does not allow to use **WHERE** clause, so we cannot filter rows while truncating.
4. DELETE activates all **delete triggers** on the table to fire. However, no triggers are fired on the truncate operation because it does not operate on individual rows.
5. DELETE performs deletion row-by-row one at a time from the table, in the order, they were processed. However, TRUNCATE operates on data pages instead of rows because it deleted entire table data at a time.
6. DELETE statement only deletes records and does not reset the **table's identity**, whereas TRUNCATE resets the identity of a particular table.
7. DELETE command require more locks and database resources because it acquires the lock on every deleted row. In contrast, TRUNCATE acquires the lock on the data page before deleting the data page; thus, it requires fewer locks and few resources.
8. DELETE statement makes an entry in the **transaction log** for each deleted row whereas, TRUNCATE records the transaction log for each data page.
9. TRUNCATE command is **faster** than the DELETE command as it deallocates the data pages instead of rows and records data pages instead of rows in transaction logs.
10. Once the record deletes by using the TRUNCATE command, we cannot recover it back. In contrast, we can recover the deleted data back which we removed from the DELETE operation.

23.Diff between having and where

A HAVING clause is like a WHERE clause, but applies only to groups as a whole (that is, to the rows in the result set representing groups), whereas the WHERE clause applies to individual rows. A query can contain both a WHERE clause and a HAVING clause. In that case:

The WHERE clause is applied first to the individual rows in the tables or table-valued objects in the Diagram pane. Only the rows that meet the conditions in the WHERE

clause are grouped.
The HAVING clause is then applied to the rows in the result set. Only the groups that meet the HAVING conditions appear in the query output. You can apply a HAVING clause only to columns that also appear in the GROUP BY clause or in an aggregate function.

24.SQL query execution order.

Clause	Function
FROM / JOIN	When you write any query, SQL starts by identifying the tables for the data retrieval and how they are connected.
WHERE	It acts as a filter; it filters the record based on the conditions specified by the users.
GROUP BY	The filtered data is grouped based on the specified condition.
HAVING	It is similar to the WHERE clause but applied after grouping the data.
SELECT	The clause selects the columns to be included in the final result.
DISTINCT	Remove the duplicate rows from the result. Once you apply this clause, you are only left with distinct records.
ORDER BY	It sorts (increasing/decreasing/A->Z/Z->A) the results based on the specified condition.
LIMIT / OFFSET	It determines the number of records to return and from where to start.

25.What are indexes? Types of Indexes and their differences.

Indexes in SQL are specialized lookup tables that are used by the database search engine to accelerate data retrieval.
In simple terms, an index in SQL is a tool used to quickly identify rows with specific column values. If there were no indexes, the SQL server would have to start with the first row and then go through the entire table until it discovers the relevant rows. This method is known as a full-table scan and can be highly inefficient for large tables.

- **Improved Query Performance**: The primary reason for using indexes is to accelerate query processing. Indexes can drastically reduce the amount of data the server needs to examine.
- **Efficient Data Access**: Indexes provide a quick way to access row data for SELECT statements. This is particularly beneficial for tables with a large number of rows.
- **Sorting and Grouping Speed**: Indexes improve the speed of data retrieval operations by providing a sorted version of the data, which is faster to process for ORDER BY and GROUP BY operations.
- **Unique Constraints**: Indexes can be used to enforce uniqueness for columns to

ensure that no two rows of a table have duplicate values in a particular column or a combination of columns.

- **Optimized Join Operations**: In databases with multiple tables, indexes improve the speed of join operations by quickly locating the joining rows in each table.

Apart from these advantages of Indexes in SQL, they have some limitations too, like:

- **Overuse of Indexes**: While indexes speed up data retrieval, they can slow down data input, through INSERT, UPDATE, and DELETE statements. Each index needs to be updated when data is modified.
- **Storage Space**: Indexes consume additional disk space.
- **Maintenance Overhead**: Indexes need to be maintained and rebuilt over time, which can add overhead to database maintenance routines.

Types of Indexes

Primary Key Index

A primary key is a field or a combination of fields in a database table that uniquely identifies each record (row) in that table. A primary key index is an automatically generated index associated with the primary key column(s) to enhance data retrieval and enforce data uniqueness.

Importance of Primary Key Index

- **Data Uniqueness:** The primary key index enforces the uniqueness constraint on the designated column(s).
  - i.e., no two records in the table can have the same values in the primary key column(s). It prevents duplicate records, ensuring data accuracy.
- **Data Retrieval Efficiency:** By creating a primary key index, the database management system (DBMS) generates a data structure that allows for rapid data retrieval. Instead of scanning the entire table, the DBMS can use the primary key index to pinpoint the exact location of a specific record, significantly improving query performance.
- **Join Operations:** Primary keys are often used in join operations, where data from multiple tables is combined. The primary key index ensures quick and efficient matching of records during these operations, reducing processing time.

Use Cases

- **Identification:** Primary keys are commonly used to identify records in a table uniquely.
  - For example, in an "Employees" table, the employee ID might serve as the primary key, allowing each employee to uniquely identify by their ID.
- **Relationships:** Primary keys are essential when establishing relationships between tables in a relational database. They serve as foreign keys in related tables, ensuring referential integrity.
- **Data Integrity:** Primary keys guarantee data integrity by preventing the insertion of duplicate records, ensuring that each record is unique.

Unique Key Index

A unique index in a relational database is a data structure that enforces the uniqueness constraint on one or more columns within a table. Its primary purpose is to ensure that values stored in the indexed column(s) are unique across all records in the table.

Role in Maintaining Unique Values:

- A unique index serves as a safeguard against duplicate data entries. It ensures that the data integrity of a table is maintained by preventing the insertion of rows with duplicate values in the indexed column(s).
- When a unique index is created on a column, the database management system

(DBMS) automatically checks for duplicate values whenever a new record is inserted or an existing record is updated in the table.

- If an insertion or update operation would result in a duplicate value in the indexed column(s), the DBMS raises an error, and the operation is rejected, thereby preventing the introduction of duplicate data.

Difference Between Primary Key Indexes and Unique Index

Index Attribute	**Primary Key**	**Unique Index**
Uniqueness Constraint	A primary key enforces uniqueness and serves as the primary identifier. It must contain non-null values and uniquely identify each row.	A unique index enforces uniqueness but does not require serving as the primary identifier. Null values are allowed as long as non-null values are unique.
Number of Columns	There can be only one primary key per table, consisting of one or more columns.	Multiple unique indexes can be created within a single table, each enforcing uniqueness on different sets of columns.
Use in Relationships	Primary keys are often used as foreign keys in related tables to establish relationships.	Unique indexes can also be used in relationships but do not have the same semantics as primary keys. They are typically used when uniqueness is needed without the requirement of being a primary identifier.

Use Cases

- **Email Addresses:** In a user database, using a unique index on the email address column ensures that each user has a unique email, preventing multiple accounts with the same email.
- **Identification Numbers:** When storing identification numbers like social security or passport numbers, a unique index ensures that no two individuals share the same identifier within the database.
- **Product SKUs:** Unique indexes can be applied to product SKU (Stock Keeping Unit) columns to prevent duplicate SKUs in an inventory database.
- **Membership IDs:** In a membership system, unique indexes on membership IDs guarantee that each member has a distinct identification number.
- **Invoice Numbers:** In financial systems, unique indexes on invoice numbers ensure that each invoice is uniquely identified, avoiding billing errors.

Clustered Index

A clustered index sorts and stores the rows of a table based on the values in one or more specified columns. Each table can have only one clustered index, and the choice of the clustering column(s) significantly impacts how data is stored and retrieved.

Importance of Clustered Index

- **Physical Data Organization:** The primary purpose of a clustered index is to physically order the data rows in the table based on the values in the indexed column(s). This arrangement allows for efficient data retrieval when queries request data in the same order as the clustered index.

- **Optimized Data Retrieval:** Clustered indexes are particularly useful for improving query performance when selecting, sorting, or filtering data based on the columns included in the clustered index. They eliminate the need for a separate data lookup process, as the data rows are already stored in the desired order.
- **Sequential Access:** When queries involve range scans or retrieving a range of data values, a clustered index is highly efficient. It allows for sequential access, reducing disk I/O operations and enhancing query speed.

Use Cases of Clustered Index

- **Primary Key:** A common use of a clustered index is to define it on the primary key column(s). This ensures that the table's data is physically ordered according to the primary key values, facilitating fast retrieval of specific records.
- **Date and Time Data:** In tables where date and time information is critical, a clustered index on a timestamp column allows for efficient retrieval of data based on chronological order.
- **Sequential Data:** For tables that store sequentially generated data, such as transaction logs or sequential invoice numbers, a clustered index can optimize the retrieval of data in chronological or sequential order.

Non-Clustered Index

A non-clustered index is a type of index used in relational databases to improve the efficiency of data retrieval operations. Unlike clustered indexes, which affect the physical order of data rows within a table, non-clustered indexes create separate data structures to allow fast access to specific data subsets. This means that non-clustered indexes do not rearrange the physical organization of data, but rather create a separate structure to facilitate quicker access to the data.

Importance of Non-Clustered Index

- **Faster Data Retrieval:** Non-clustered indexes significantly improve query performance by allowing the database management system (DBMS) to quickly locate and retrieve specific data rows based on the indexed column(s).
- **Reduced Disk I/O:** Non-clustered indexes reduce the need for full table scans when querying data. This leads to fewer disk input/output (I/O) operations, resulting in faster query execution.
- **Support for Multiple Indexes:** Unlike clustered indexes, which limit a table to one, non-clustered indexes can be created on multiple columns, enabling efficient retrieval for various query patterns.

Difference Between Clustered and Non-Clustered Index

Index Type	**Non-Clustered Index**	**Clustered Index**
Physical Order	Does not determine the physical order of data rows.	Determines the physical order of data rows based on indexed column(s).
Data Structure	Creates a separate data structure containing index entries with pointers to the corresponding data rows.	Organizes the actual data rows in the table in the specified order.
Multiple Indexes	Allows for multiple non-clustered indexes on a single table.	Restricts a table to having only one clustered index.
Query	Suitable for optimizing query	Ideal for queries that frequently

Optimization	performance when the query does not align with the physical data order.	retrieve data in the same order as the clustering column(s).

SQL Indexes - Types, Uses, Advantages, Disadvantages, and Scenarios

Index Type	**Use**	**Advantages**	**Disadvantages**	**Ideal Scenarios**
Primary Key Index	Automatically created with the primary key to enforce uniqueness	Fast data retrieval; Ensures data integrity	Additional storage; Slower insert/update operations	Unique identifier for each row (e.g., UserID)
Unique Index	Enforces uniqueness on a column not part of the primary key	Prevents duplicate values; Improves search performance	Slower write operations; Additional storage requirement	Columns requiring unique data but not suitable as a primary key (e.g., Email)
Clustered Index	Determines the physical storage order of data in the table	Fast data retrieval for range queries; Efficient use of disk space	Only one per table; Update operations can be slow due to reordering	Columns frequently used in sorting and range queries (e.g., Dates)
Non-Clustered Index	Provides a separate structure from the data rows and includes a pointer	Faster access than table scan; Multiple non-clustered indexes allowed per table	Increased storage; Slower write operations due to index updates	Frequently searched fields not in clustered index (e.g., FirstName)
Composite Index	Index on two or more columns	Improves performance on queries involving multiple columns	More complex, Increased storage; Slower writes	Multi-column searches and sorting (e.g., FirstName, LastName)
Full-Text Index	Used for full-text searches in text data	Facilitates complex queries on text data; Faster than LIKE searches	Takes up significant storage space; Specialized use-case	Large text fields (e.g., Product Descriptions, Articles)
Bitmap Index	Efficient for columns with a low cardinality (few unique values)	Small storage space; Fast for read-intensive tasks	Not suitable for frequently changing data; Performance issues with high cardinality data	Columns with limited unique entries (e.g., Gender, Marital Status)

Spatial Index	For indexing spatial data types	Improves performance for queries involving spatial data	Specific use-case; Additional complexity	Geographical data, location-based queries (e.g., Maps, Regions)

26.What is surrogate key? Give example where you used it and how.

A surrogate key is a key which does not have any contextual or business meaning. It is manufactured “artificially” and only for the purposes of data analysis. The most frequently used version of a surrogate key is an increasing sequential integer or “counter” value (i.e. 1, 2, 3). Surrogate keys can also include the current system date/time stamp, or a random alphanumeric string.