df\_sample['Discounted Price']**=**df\_sample['Sales']**.**apply(**lambda** x:0.85**\***x **if** x**>**200 **else** x)

df\_sample**.**head(10)

df\_sample['Customer Name Length']**=**df\_sample['Customer Name']**.**apply(len)

df\_sample**.**head(10)

**def** categorize\_sales(price):

**if** price **<** 50:

**return** "Low"

**elif** price **<** 200:

**return** "Medium"

**else**:

**return** "High"

df\_sample['Sales Price Category']**=**df\_sample['Sales']**.**apply(categorize\_sales)

df\_sample**.**head(10)

df\_sample**.**sort\_values(by**=**['State','Sales'])

df\_sample**=**df[['Customer Name','State','Sales','Quantity']]**.**sample(n**=**15)

df\_sample

df['Customer Name']**.**value\_counts()[:10]

df\_cat1 **=** pd**.**concat([df\_1,df\_2,df\_3], axis**=**0)

df\_cat1

df\_missing[['Customer','Product']]**.**fillna('FILL')

print(pd**.**DataFrame(df\_subset**.**groupby('State')**.**describe()**.**loc['California'])**.**transpose())

df\_subset[(df\_subset['State']**!=**'California') **&** (df\_subset['Sales']**>**100)]

df['State']**.**unique()

df['State']**.**nunique()

df3[df3['Flavanoids']**>=**3.4975][['Ash','Alcohol','Magnesium']]**.**mean()

SELECT CITY, LENGTH(CITY) FROM STATION

ORDER BY LENGTH(CITY), CITY ASC

LIMIT 1;

SELECT CITY, LENGTH(CITY) FROM STATION

ORDER BY LENGTH(CITY) DESC, CITY ASC

LIMIT 1;

SELECT name FROM students WHERE marks>75 ORDER BY RIGHT(name,3) ,ID;

SELECT

CASE

WHEN A=B AND B=C THEN 'Equilateral'

WHEN (A=B OR B=C OR C=A) AND A+B > C THEN 'Isosceles'

WHEN A+B > C THEN 'Scalene'

ELSE 'Not A Triangle'

END AS 'TRIANGLE'

FROM TRIANGLES

SELECT CONCAT(Name,'',CONCAT('(','',CONCAT(LEFT(Occupation,1),'', ')')) )

FROM OCCUPATIONS

ORDER BY Name;

SELECT CONCAT('There are a total of ',COUNT(Occupation),' ',LOWER(occupation),'s','.')

FROM OCCUPATIONS GROUP BY Occupation ORDER BY COUNT(Occupation);

**What they do**

RANK and DENSE\_RANK are used to order values and assign them numbers depending on where they fall in relation to one another.

For example, let’s say you have 3 students with 3 different test scores- one student received a 100, another received an 85, and the last received a 72. These functions will assign a 1, 2, or 3 to each student depending on the order you wish to rank them.

SELECT student\_name, RANK() OVER(ORDER BY grades DESC) AS grade\_ranking

**ORDER BY** specifies the column whose values you wish to rank. In the example earlier, grades would be specified after ORDER BY. You can order by descending or ascending values.

<https://towardsdatascience.com/how-to-use-sql-rank-and-dense-rank-functions-7c3ebf84b4e8>

# Differences between RANK and DENSE\_RANK

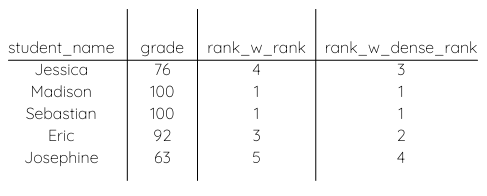
The difference between these two functions comes down to how they handle identical values. Let’s say we have two students who have the same grade; both scored 90s on their math test.

RANK and DENSE\_RANK will assign the grades the same rank depending on how they fall compared to the other values. However, RANK will then skip the next available ranking value whereas DENSE\_RANK would still use the next chronological ranking value.

So with RANK, if the two 90s are given a ranking of 2, the next lowest value would be assigned a rank of 4, skipping over 3. With DENSE\_RANK, the next lowest value would be assigned a rank of 3, not skipping over any values.

Let’s compare the outcomes of both of these functions.

SELECT student\_name, RANK() OVER(ORDER BY grades DESC) AS rank\_w\_rank, DENSE\_RANK() OVER(ORDER BY grades DESC) AS rank\_w\_dense\_rank



SELECT

N,

CASE

WHEN P is NULL THEN 'Root'

WHEN N IN (SELECT P FROM BST) THEN 'Inner'

ELSE 'Leaf'

END AS node\_type

FROM

BST

ORDER BY

N;

SELECT SUM(CITY.POPULATION)

FROM CITY

JOIN COUNTRY

ON CITY.COUNTRYCODE = COUNTRY.CODE

WHERE COUNTRY.CONTINENT = 'ASIA';

select cc.continent, floor(avg(c.population))

from city c

inner join country cc

on c.countrycode = cc.code

group by cc.continent;

### **Q4. Write a query to retrieve the first four characters of  EmpLname from the EmployeeInfo table.**

|  |  |
| --- | --- |
| 1 | **SELECT** SUBSTRING(EmpLname, 1, 4) **FROM** EmployeeInfo; |

**CREATE** **TABLE** NewTable **AS** **SELECT** \* **FROM** EmployeeInfo;

**SELECT** **TOP** N \* **FROM** EmployeePosition **ORDER** **BY** Salary **DESC**;

**SELECT** CONCAT(EmpFname, ' ', EmpLname) **AS** 'FullName' **FROM** EmployeeInfo;

**SELECT** \* **FROM** EmployeeInfo **WHERE** EmpFname NOT IN ('Sanjay','Sonia');

**SELECT** E.EmpFname, E.EmpLname, P.EmpPosition

**FROM** EmployeeInfo E **INNER** JOIN EmployeePosition P **ON**

E.EmpID = P.EmpID AND P.EmpPosition IN ('Manager');

SELECT companies.permalink AS companies\_permalink,

companies.name AS companies\_name,

acquisitions.company\_permalink AS acquisitions\_permalink,

acquisitions.acquired\_at AS acquired\_date

FROM tutorial.crunchbase\_companies companies

LEFT JOIN tutorial.crunchbase\_acquisitions acquisitions

ON companies.permalink = acquisitions.company\_permalink

WHERE acquisitions.company\_permalink != '/company/1000memories'

OR acquisitions.company\_permalink IS NULL

ORDER BY 1

#### Q-25. Write an SQL query to fetch duplicate records having matching data in some fields of a table.

**Ans.**

The required query is:

SELECT WORKER\_TITLE, AFFECTED\_FROM, COUNT(\*)

FROM Title

GROUP BY WORKER\_TITLE, AFFECTED\_FROM

HAVING COUNT(\*) > 1;

SELECT getdate();

SELECT TOP 1 Salary

FROM (

SELECT DISTINCT TOP n Salary

FROM Worker

ORDER BY Salary DESC

)

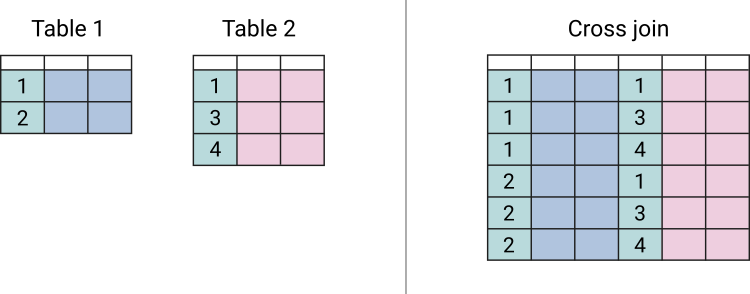
ORDER BY Salary ASC;

<https://learnsql.com/blog/sql-join-interview-questions-with-answers/>

### CROSS JOIN

An SQL CROSS JOIN is used when you need to find out all the possibilities of combining two tables, where the result set includes every row from each contributing table. The CROSS JOIN clause returns the Cartesian product of rows from the joined tables.

The diagram below is a good illustration of how the rows are combined:



### SQL NATURAL JOIN

A NATURAL JOIN is a type of JOIN that combines tables based on columns with the same name and data type. When you use the NATURAL JOIN clause, it creates an implicit JOIN clause for you based on the common columns in the two tables being joined.

Common columns are columns that have the same name in both tables. There is no need to specify the column names to join. The resulting table will not contain any repeated columns.

The syntax for a NATURAL JOIN is simple:

|  |
| --- |
| SELECT \* FROM employees  NATURAL JOIN departments; |

## 4. What is an OUTER JOIN?

With an SQL OUTER JOIN, unmatched rows in one or both tables can be returned. There are several variations of the OUTER JOIN clause, some of which we have covered already in this article. Here are the common types of OUTER JOIN clauses:

* LEFT OUTER JOIN
* RIGHT OUTER JOIN
* FULL OUTER JOIN

LEFT JOIN is a synonym for LEFT OUTER JOIN. The functionality of both is identical. This may be one of the SQL JOIN interview questions you are asked! The same can be said for RIGHT JOIN and RIGHT OUTER JOIN, and FULL JOIN and FULL OUTER JOIN.

#### Q-50. Write an SQL query to fetch the names of workers who earn the highest salary.

**Ans.**

The required query is:

SELECT FIRST\_NAME, SALARY from Worker WHERE SALARY=(SELECT max(SALARY) from Worker);

**CREATE** **TABLE** Students ( /\* Create table with a single field as primary key \*/

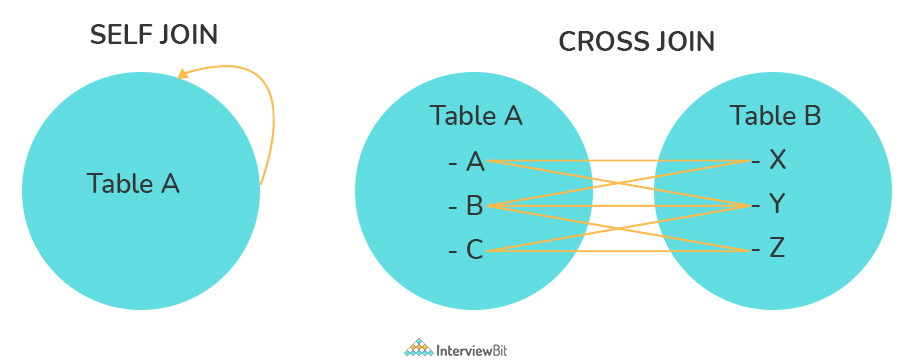
ID INT **NOT** **NULL**

Name VARCHAR(255)

**PRIMARY** KEY (ID)

);

Cross join can be defined as a cartesian product of the two tables included in the join. The table after join contains the same number of rows as in the cross-product of the number of rows in the two tables. If a WHERE clause is used in cross join then the query will work like an INNER JOIN.



Constraints are used to specify the rules concerning data in the table. It can be applied for single or multiple fields in an SQL table during the creation of the table or after creating using the ALTER TABLE command. The constraints are:

* **NOT NULL** - Restricts NULL value from being inserted into a column.
* **CHECK** - Verifies that all values in a field satisfy a condition.
* **DEFAULT** - Automatically assigns a default value if no value has been specified for the field.
* **UNIQUE** - Ensures unique values to be inserted into the field.
* **INDEX** - Indexes a field providing faster retrieval of records.
* **PRIMARY KEY** - Uniquely identifies each record in a table.
* **FOREIGN KEY** - Ensures referential integrity for a record in another table

**CREATE** **TABLE** Students ( /\* Create table with foreign key - Way 1 \*/

ID INT **NOT** **NULL**

Name VARCHAR(255)

LibraryID INT

**PRIMARY** KEY (ID)

**FOREIGN** KEY (Library\_ID) **REFERENCES** Library(LibraryID)

);

A database index is a data structure that provides a quick lookup of data in a column or columns of a table. It enhances the speed of operations accessing data from a database table at the cost of additional writes and memory to maintain the index data structure.

**CREATE** INDEX index\_name /\* Create Index \*/

**ON** table\_name (column\_1, column\_2);

**DROP** INDEX index\_name; /\* Drop Index \*/

Clustered indexes are indexes whose order of the rows in the database corresponds to the order of the rows in the index. This is why only one clustered index can exist in a given table, whereas, multiple non-clustered indexes can exist in the table.

The only difference between clustered and non-clustered indexes is that the database manager attempts to keep the data in the database in the same order as the corresponding keys appear in the clustered index.

Clustering indexes can improve the performance of most query operations because they provide a linear-access path to data stored in the database.

### 27. What is Normalization?

Normalization represents the way of organizing structured data in the database efficiently. It includes the creation of tables, establishing relationships between them, and defining rules for those relationships. Inconsistency and redundancy can be kept in check based on these rules, hence, adding flexibility to the database.

### 28. What is Denormalization?

Denormalization is the inverse process of normalization, where the normalized schema is converted into a schema that has redundant information. The performance is improved by using redundancy and keeping the redundant data consistent. The reason for performing denormalization is the overheads produced in the query processor by an over-normalized structure.

**DELETE** statement is used to delete rows from a table.

**TRUNCATE** command is used to delete all the rows from the table and free the space containing the table.

### 33. What are Aggregate and Scalar functions?

An aggregate function performs operations on a collection of values to return a single scalar value. Aggregate functions are often used with the GROUP BY and HAVING clauses of the SELECT statement. Following are the widely used SQL aggregate functions:

* **AVG()** - Calculates the mean of a collection of values.
* **COUNT()** - Counts the total number of records in a specific table or view.
* **MIN()** - Calculates the minimum of a collection of values.
* **MAX()** - Calculates the maximum of a collection of values.
* **SUM()** - Calculates the sum of a collection of values.
* **FIRST()** - Fetches the first element in a collection of values.
* **LAST()** - Fetches the last element in a collection of values.

**Note:** All aggregate functions described above ignore NULL values except for the COUNT function.

A scalar function returns a single value based on the input value. Following are the widely used SQL scalar functions:

* **LEN()** - Calculates the total length of the given field (column).
* **UCASE()** - Converts a collection of string values to uppercase characters.
* **LCASE()** - Converts a collection of string values to lowercase characters.
* **MID()** - Extracts substrings from a collection of string values in a table.
* **CONCAT()** - Concatenates two or more strings.
* **RAND()** - Generates a random collection of numbers of a given length.
* **ROUND()** - Calculates the round-off integer value for a numeric field (or decimal point values).
* **NOW()** - Returns the current date & time.
* **FORMAT()** - Sets the format to display a collection of values
* Q7. Write an SQL query to fetch duplicate records from Patients, without considering the primary key.
* **SELECT** PatientName, DoctorID, RegDate, State, **COUNT**(\*)
* **FROM** Patients
* **GROUP** **BY** PatientName, DoctorID, RegDate, State
* **HAVING** **COUNT**(\*) > 1;

Q28. Write a query to fetch PatientIDs  which are present in:

* Both tables
* One of the table. Let us say, patients present in Patients and not in the PatientsCheckup table.

–Present **IN** **BOTH** **TABLES**

**SELECT** PatientId **FROM** Patients

**WHERE** PatientId **IN**

(**SELECT** PatientId **FROM** PatientsCheckup);

SELECT ROUND(AVG(population),0) FROM CITY;