SQL JOINS: The purpose of a join is to combine the data across tables.

**Inner Join:**

* This will display all the records that have matched from both tables.

SELECT customers.Name, orders.order\_id, orders.order\_date

FROM customers

INNER JOIN orders

ON customers.id = orders.customer\_id;

**Outer Join:**

* Outer join gives the non-matching records along with matching records.

**Left Outer Join:**

* Left outer join returns all records from left table, at same time, it brings all matching rows from right table and NULL appears in right side when no matching row exists.

SELECT customers.Name, orders.order\_id, orders.order\_date

FROM customers

LEFT JOIN orders

ON customers.id = orders.customer\_id;

**Right Outer Join**:

* Right outer join returns all records from right table, at the same time, it brings all matching rows from left table and NULL appears in left side when no matching row exists.

SELECT customers.Name, orders.order\_id, orders.order\_date

FROM customers

RIGHT JOIN orders

ON customers.id = orders.customer\_id;

**Full Outer Join:**

* Returns all records from both tables. If no match, NULL is returned.

SELECT customers.Name, orders.order\_id, orders.order\_date

FROM customers

FULL OUTER JOIN orders

ON customers.id = orders.customer\_id;

**Self-Join:**

* Joins a table with itself to find relationships within the same table.

SELECT e1.name AS Employee, e2.name AS Manager

FROM employees e1

LEFT JOIN employees e2

ON e1.manager\_id = e2.employee\_id;

**Natural Join:**

* Natural join compares all the common columns.

**Cross Join:**

* Returns the Cartesian product of both tables (every row from the first table matches every row from the second). If Employees has 4 rows and Departments has 3 rows, then Total Rows = 4 x 3 = 12 (All Possible Combinations)

SELECT customers.Name, orders.order\_id, orders.order\_date

FROM customers

CROSS JOIN orders

A diagram of a group of circles

AI-generated content may be incorrect.

**Sub Queries:**

* Nesting of queries, i.e., a query which is inside another query. Here the outer query is called the parent query, which gets a result from the inner query. So, the inner query gets executed first and based on its result the outer query gets executed.

**Find the Highest salary from Employee Table**

SELECT \*

FROM Employee Emp1

WHERE (1) = (

SELECT COUNT (DISTINCT (Emp2.Salary))

FROM Employee Emp2

WHERE Emp2.Salary > Emp1.Salary)

Query for identifying duplicate rows

**SELECT CARID, CARTYPE, CARCOLOR from CARS GROUP BY CARID, CARTYPE, CARCOLOR HAVING COUNT (\*)>1**

**Difference between Truncate and Delete in SQL**

**–> DELETE:**  
1. Removes **Some** or **All** rows from a table.  
  
2. A **WHERE** clause can be used to remove some rows. If no WHERE condition is specified, all rows will be removed.  
  
3. Causes all DELETE **triggers** on the table to fire.  
  
4. It removes rows **row-by-row** one at a time and records an entry in the Transaction logs, thus is slower than TRUNCATE.  
  
5. Every deleted row in locked, thus it requires more number of **locks** and database resources.  
  
6. This is a **DML** command as it is just used to manipulate/modify the table data. It does not change any property of a table.  
   
**–> TRUNCATE:**   
1. Removes **All** rows from a table.  
  
2. Does not require a **WHERE** clause, so you cannot filter rows while Truncating.   
3. **No Triggers** are fired on this operation because it does not operate on individual rows.  
  
4. It de-allocates **Data Pages** instead of Rows and records Data Pages instead of Rows in Transaction logs, thus is faster than DELETE.  
  
5. While de-allocating Pages it locks Pages and not Rows, thus it requires less number of **locks** and few resources.  
  
6. TRUNCATE is not possible when a table:  
a. is reference by a **Foreign Key** or tables used in replication or with Indexed views.  
b. participates in an Indexed/Materialized View.  
c. published by using Transactional/Merge replication.  
  
7. This is a **DDL** command as it **resets IDENTITY** columns, de-allocates Data Pages and empty them for use of other objects in the database.

**Difference between Clustered and Non-clustered Indexes in SQL**  
1) Clustered Index physically sort all rows while Non clustered Index doesn't.  
2) In SQL one table can only have one Clustered Index, but there is no such restriction on Non-Clustered Index.  
3) In many relational databases Clustered Index is automatically created on primary key column.

**Data Modeling**

* Data modeling is the process of designing structured representation of data in a database.

**Types of Data Models**

* **Conceptual Model**: High-level business structure
* **Logical Model**: Defines tables, columns and relationships
* **Physical Model**: Actual Database Schema (Indexing, Partitions)  
    
  **ACID properties in Transaction**
* **Atomicity**: which monitors either a transaction is completely done or nothing has been done.
* **Consistency**: through which valid data gets into database
* **Isolation**: If you are in the middle of the transactions other processes wont able to see the data until the transaction is completed
* **Durability**: Once the transaction is completed, even if the system crashes, there should be a way to get back the data

**Primary Key**

* Primary key uniquely identify a record in the table.
* Primary Key can't accept null values.
* By default, Primary key is clustered index and data in the database table is physically organized in the sequence of clustered index.
* We can have only one Primary key in a table.

**Foreign Key**

* Foreign key is a field in the table that is primary key in another table.
* Foreign key can accept multiple null value.
* Foreign key do not automatically create an index, clustered or non-clustered. You can manually create an index on foreign key.
* We can have more than one foreign key in a table.

**Unique Key**

* Unique key can accept only one null value.
* By default, Unique key is a unique non-clustered index.
* We can have more than one unique key in a table.

**Surrogate Key**

* Surrogate key is a unique identifier (usually auto incremented IDs) that has no business meaning but it is used as a Primary Key

**Candidate Key**

* Candidate Key can be any column or a combination of columns that can qualify as unique key in database.
* There can be multiple Candidate Keys in one table.
* Each Candidate Key can qualify as Primary Key.

**Merge Statement**

* The **merge** statement combines **INSERT, UPDATE** and **DELETE** Operations into a single query based on conditions
* It helps in synchronizing tables, upserting and efficient bulk operations

**Difference between JOINS and UNIONS**

* **JOINS** combines columns from multiple tables based on a condition (ON)
* **UNIONS** stacks the results of two queries vertically (must have the same number of columns)
* **JOINS** retains all matching values, while **UNIONS** remove duplicates (unless **UNION ALL** used)

**Types of Indexes**

**🔹 1. Primary Key Index**

* Automatically created when a **PRIMARY KEY** is defined.
* Ensures **uniqueness** of values in the column.

✔️ **Automatically creates an index on id**.

**🔹 2. Unique Index**

* Ensures **all values in a column are unique**.
* Prevents duplicate entries.
* CREATE UNIQUE INDEX idx\_unique\_email ON employees (email);

**🔹 3. Composite Index (Multi-Column Index)**

* Indexes **multiple columns** together.
* Useful for **queries filtering by multiple columns**.
* CREATE INDEX idx\_composite ON employees (department\_id, salary);

**🔹 4. Clustered Index**

* **Physically reorders data** in a table based on the index.
* Only **one clustered index per table**.
* CREATE CLUSTERED INDEX idx\_clustered ON employees (id);

✔️ Improves performance for **range queries** and **sorting**.

**🔹 5. Non-Clustered Index**

* Stores index **separately from actual data**.
* **Multiple non-clustered indexes** per table.
* CREATE NONCLUSTERED INDEX idx\_nonclustered ON employees (name);

✔️ Improves **search performance** but requires extra storage.

**🔹 6. Full-Text Index**

* Used for **fast text searches** in large text fields.
* Works with CONTAINS() and MATCH() functions.
* CREATE FULLTEXT INDEX idx\_fulltext ON employees(name);

✔️ Efficient for **searching large text fields**.

**🔹 7. Partial Index (Filtered Index)**

* Indexes only **a subset of rows** based on a condition.
* CREATE INDEX idx\_partial ON employees (salary) WHERE salary > 50000;

✔️ Saves space and improves performance for **specific queries**.

**🔹 8. Covering Index**

* Index **includes all columns** needed for a query.
* Avoids accessing the **main table**, improving performance.
* CREATE INDEX idx\_covering ON employees (department\_id, salary, name);

**🔹 9. Hash Index**

* Uses **hashing** for **fast lookups**.
* Best for **exact match queries**, but **not for range queries**.
* CREATE INDEX idx\_hash ON employees USING HASH (id);

**🔹 10. Spatial Index (GIS Index)**

* Optimized for **geospatial data**.
* ✔️ Used in **location-based queries**.

JDBC transaction isolation levels

**TRANSACTION\_NONE**

     This is a special constant indicating that the JDBC driver does not support transactions.

**TRANSACTION\_READ\_UNCOMMITTED**

    This level allows transactions to see uncommitted changes to the data. All database anomalies are possible at this level.

**TRANSACTION\_READ\_COMMITTED**

    This level means that any changes made inside a transaction are not visible outside it until the transaction is committed. This prevents dirty reads from being possible.

**TRANSACTION\_REPEATABLE\_READ**

    This level means that rows that are read retain locks so that another transaction cannot change them when the transaction is not completed. This disallows dirty reads and nonrepeatable reads. Phantom read are still possible.

**TRANSACTION\_SERIALIZABLE**

    Tables are locked for the transaction so that WHERE conditions cannot be changed by other transactions that add values to or remove values from a table. This prevents all types of database anomalies.

### Transaction Propagation

[MANDATORY](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#MANDATORY)  
          Support a current transaction, throw an exception if none exists.

[NESTED](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#NESTED)  
          Execute within a nested transaction if a current transaction exists, behave like PROPAGATION\_REQUIRED else.

[NEVER](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#NEVER)  
          Execute non-transactionally, throw an exception if a transaction exists.

[NOT\_SUPPORTED](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#NOT_SUPPORTED)  
          Execute non-transactionally, suspend the current transaction if one exists.

[REQUIRED](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#REQUIRED)  
          Support a current transaction, create a new one if none exists.

[REQUIRES\_NEW](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#REQUIRES_NEW)  
          Create a new transaction, suspend the current transaction if one exists.

[SUPPORTS](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#SUPPORTS)  
          Support a current transaction, execute non-transactionally if none exists.

**Nested Subqueries**  
A subquery is nested when you are having a subquery in the where or having clause of another subquery.  
  
Get the result of all the students who are enrolled in the same course as the student with ROLLNO 12.

Select \*

From result

where rollno in (select rollno

from student

where courseid = (select coursed from student where rollno = 12));

The innermost subquery will be executed first and then based on its result the next subquery will be executed and based on that result the outer query will be executed. The levels to which you can do the nesting is implementation dependent.  
  
**Correlated Subquery**  
  
A **Correlated Subquery** is one that is executed after the outer query is executed. So correlated subqueries take an approach opposite to that of normal subqueries. The correlated subquery execution is as follows:

-The outer query receives a row.  
-For each candidate row of the outer query, the subquery (the correlated subquery) is executed once.  
-The results of the correlated subquery are used to determine whether the candidate row should be part of the result set.  
-The process is repeated for all rows.

*Correlated Subqueries* differ from the normal subqueries in that the nested SELECT statement refers back to the table in the first SELECT statement.  
  
To find out the names of all the students who appeared in more than three papers of their opted course, the SQL will be

Select name

from student A

Where 3 < (select count (\*)

from result b

where b.rollno = a.rollno);

In other words, a correlated subquery is one whose value depends upon some variable that receives its value in some outer query. A non-correlated subquery as said before is evaluted in a bottom-to-up manner, i.e. the inner most query is evaluated first. But a correlated subquery is resolved in a top-to-bottom fashion. The top most query is analyzed and based on that result the next query is initiated. Such a subquery has to be evaluated repeatedly, once for each value of the variable in question, instead of once and for all.

**Normalization**:

Normalization is the process of organizing data in a database to minimize redundancy and improve data integrity. The normal forms are:

* **1NF (First Normal Form)**: Ensures that each column contains atomic values and that each column contains values of a single type.
* **2NF (Second Normal Form)**: Meets all requirements of 1NF, and all non-key attributes are fully functionally dependent on the primary key.
* **3NF (Third Normal Form)**: Meets all requirements of 2NF, and all attributes are only dependent on the primary key (no transitive dependency).
* **BCNF (Boyce-Codd Normal Form)**: A stricter version of 3NF where every determinant is a candidate key.

**Retrieve the top 3 employees by salary for each department.**

SELECT department\_id, employee\_name, salary FROM ( SELECT department\_id, employee\_name, salary, DENSE\_RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS rank FROM employees ) ranked WHERE rank <= 3;

**To fetch the products that are present in the product table but do not have corresponding entries in the category table, you can use a LEFT JOIN along with the WHERE clause to filter out the NULL values in the category table.**

SELECT p.product\_id

FROM product p

LEFT JOIN category c ON p.product\_id = c.product\_id

WHERE c.product\_id IS NULL;

**Select first highest salary of employee:**

SELECT employee\_name, salary FROM employees ORDER BY salary DESC LIMIT 1;

Or

SELECT employee\_name, salary

FROM employees

WHERE salary = (SELECT MAX(salary) FROM employees);

**Select second highest salary of employee:**

SELECT distinct employee\_name, salary FROM employees ORDER BY salary DESC LIMIT 1 OFFSET 1;

Or

SELECT employee\_name, salary

FROM employees

WHERE salary = (

SELECT MAX(salary)

FROM employees

WHERE salary < (SELECT MAX(salary) FROM employees)

);

**Fetch First 4 characters of EmpName**

Select LEFT(empName, 4) empName from emp;

**Retrieve Concatenate first name and last name**

Select CONCAT( firstname, ‘ ‘, lastName) full name from employee;

**Fetch Top N Records**

SELECT \* from employee order by salary desc limit N;

**Fetch common records from two tables**

Select \* from tableA

INTERSECT

Select \* from tableB

OR

SELECT A.\* FROM tableA a INNER JOIN tableB b ON a.id = b.id

**1️ Find Second Highest Salary**

SELECT MAX(salary) AS second\_highest\_salary

FROM employees

WHERE salary < (SELECT MAX(salary) FROM employees);

**2️ Find Nth Highest Salary**

SELECT DISTINCT salary FROM employees ORDER BY salary DESC LIMIT 1 OFFSET N-1;

**3️ Find Top 3 Highest Salaries from Each Department**

SELECT department\_id, employee\_name, salary

FROM (

SELECT department\_id, employee\_name, salary,

RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS rnk

FROM employees

) ranked

WHERE rnk <= 3;

**4️ Find Duplicate Records in a Table**

SELECT column\_name, COUNT(\*)

FROM table\_name

GROUP BY column\_name

HAVING COUNT(\*) > 1;

**5️ Find Employees Who Earn More Than Their Managers**

SELECT e.name, e.salary

FROM employees e

JOIN employees m ON e.manager\_id = m.id

WHERE e.salary > m.salary;

**6️ Find Common Records Between Two Tables**

SELECT \* FROM table1

INTERSECT

SELECT \* FROM table2;

**7️ Find Employees with the Highest Salary in Each Department**

SELECT department\_id, employee\_name, salary

FROM (

SELECT department\_id, employee\_name, salary,

RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS rnk

FROM employees

) ranked

WHERE rnk = 1;

**8️ Find Missing Numbers in a Sequence**

SELECT t1.number + 1 AS missing\_number

FROM numbers t1

LEFT JOIN numbers t2 ON t1.number + 1 = t2.number

WHERE t2.number IS NULL;

**Find the Second Highest Salary**

SELECT MAX(salary) AS SecondHighestSalary

FROM employees

WHERE salary < (SELECT MAX(salary) FROM employees);

**Find the Third Highest Salary**

SELECT DISTINCT salary

FROM employees

ORDER BY salary DESC

LIMIT 1 OFFSET 2;

**Find the Department with the Highest Number of Employees**

SELECT department\_id, COUNT(\*) AS employee\_count

FROM employees

GROUP BY department\_id

ORDER BY employee\_count DESC

LIMIT 1;

**Find Employees Who Earn More Than Their Manager**

SELECT e1.name AS Employee, e1.salary, e2.name AS Manager, e2.salary AS ManagerSalary

FROM employees e1

JOIN employees e2 ON e1.manager\_id = e2.id

WHERE e1.salary > e2.salary;

**Retrieve Duplicate Records from a Table**

SELECT name, COUNT(\*)

FROM employees

GROUP BY name

HAVING COUNT(\*) > 1;

**Find Employees Who Don't Have a Manager**

SELECT name FROM employees WHERE manager\_id IS NULL;

**Find the Employees Who Joined in the Last 30 Days**

SELECT \* FROM employees

WHERE joining\_date >= CURRENT\_DATE - INTERVAL '30 days';

**Find the Top 3 Salaries in Each Department**

SELECT department\_id, name, salary

FROM (

SELECT \*, RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS rank

FROM employees

) ranked

WHERE rank <= 3;

**Find Employees Who Share the Same Salary**

SELECT e1.name, e1.salary, e2.name AS AnotherEmployee

FROM employees e1

JOIN employees e2 ON e1.salary = e2.salary AND e1.id <> e2.id;

**Find Employees in Each Department with the Highest Salary**

SELECT department\_id, name, salary

FROM employees

WHERE (department\_id, salary) IN (

SELECT department\_id, MAX(salary)

FROM employees

GROUP BY department\_id

);

**Find the Number of Employees in Each Department**

SELECT department\_id, COUNT(\*) AS employee\_count

FROM employees

GROUP BY department\_id;

**Find Employees Who Joined Between Two Dates**

SELECT \* FROM employees

WHERE joining\_date BETWEEN '2023-01-01' AND '2023-12-31';

**Find the Average Salary of Each Department**

SELECT department\_id, AVG(salary) AS avg\_salary

FROM employees

GROUP BY department\_id;

**Find the Employees Who Have the Same First Name**

SELECT first\_name, COUNT(\*)

FROM employees

GROUP BY first\_name

HAVING COUNT(\*) > 1;

**Retrieve the Employees Who Have the Maximum Experience**

SELECT \* FROM employees

ORDER BY joining\_date ASC LIMIT 1;

**Find the Total Salary Paid to Each Department**

SELECT department\_id, SUM(salary) AS total\_salary

FROM employees

GROUP BY department\_id;

**Find the Employees Who Have the Same Manager**

SELECT manager\_id, COUNT(\*)

FROM employees

GROUP BY manager\_id

HAVING COUNT(\*) > 1;

**Write a Query to Find the Median Salary**

SELECT salary

FROM employees

ORDER BY salary

LIMIT 1 OFFSET (SELECT COUNT(\*) FROM employees) / 2;

**Find the Most Frequently Used Department**

SELECT department\_id, COUNT(\*) AS employee\_count

FROM employees

GROUP BY department\_id

ORDER BY employee\_count DESC

LIMIT 1;

**Find Employees Who Have Never Been Assigned to a Department**

SELECT \* FROM employees

WHERE department\_id IS NULL;