https://www.sanfoundry.com/1000-sql-questions-answers/

In SQL, we can insert and delete rows in a view. We can perform insert and update in an updatable view only. Read-only views are not updatable.

PL in PL/SQL stands for Procedural Language. It is a block structured language that enables developers to combine the power of SQL with procedural language.

A trigger is automatically executed by the Oracle server whenever a specified event occurs. A user can create and store a trigger and specify when to execute it but the its execution is done by the server automatically.

The SQL LIKE operator is used in a WHERE clause to search for a specified pattern in a column

We can update only a single table at a time using UPDATE

A Database table consists of columns and rows. These columns are known as Attributes and Fields, rows are known as Tuples and Records.

The data in RDBMS is stored in database objects called tables, which is a collection of related data entries.

The SELECT statement is used to retrieve data from a database. The data retrieved by a SELECT clause is stored in a result table, called as result-set

Select corresponds to Projection, from corresponds to Selection and where corresponds to cartesian product operations of Relational algebra.

In SQL, the DISTINCT clause doesn’t ignore NULL values. So when using the DISTINCT clause in SQL statement, our result set will include NULL as a distinct value.

SQL Distinct clause treats NULL as a unique value.

HAVING is used to filter groups. WHERE is used to filter rows. It filters out the unwanted data records. SELECT is used to filter columns.

Where clause can be used along with Select, Delete Update commands. But it cannot be used with Drop command to delete a table.

<> denotes not equal. In some versions of SQL “!=” is also used for not equal.

 == is not a valid operator in SQL and it is not used with where clause.

Not all the database systems supports SELECT TOP clause. It can be used in SQL. Mysql supports the LIMIT clause to fetch a limited number of records from the database table.

SELECT TOP clause is used in SQL to fetch limited records from a table. MYSQL supports LIMIT clause and ORACLE uses ROWNUM clause to select limited number of rows from a database table.

**SELECT** column\_list **FROM** **TABLE\_NAME** **WHERE** ROWNUM<=**VALUE**;

Select top clause is used to select records from the result table not from the original database table. Result table contains the records of database table which satisfies the conditions provided in the where clause of select statement.

"Funny Jokes Will Grab Happy Smiles Often"  
F → FROM (Tables and Joins)  
J → JOIN (Combine data sources, part of FROM)  
W → WHERE (Filter rows)  
G → GROUP BY (Group data for aggregation)  
H → HAVING (Filter aggregated groups)  
S → SELECT (Choose columns, compute expressions)  
O → ORDER BY (Sort results)

Mnemonic for Logical Execution Order:  
  
First We Gather Hopeful Solutions Out Loud  
(FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT)  
  
note: Joins are part of the FROM clause in SQL's logical execution order, so they are not explicitly listed as a separate step.  
  
Written Order:  
SELECT  
FROM  
JOIN (with ON conditions)  
WHERE  
GROUP BY  
HAVING  
ORDER BY  
LIMIT

Percent clause is used with SELECT TOP to select given percentage of records from the result table. Syntax for such statement is:

**SELECT** top **NUMBER** Percent column\_list **FROM** **TABLE\_NAME** **WHERE** condition(s);

Select top 30 percent \* from student;

**SELECT** top **VALUE** Percent column\_list **FROM** **TABLE\_NAME** **WHERE** condition(s);

NULL:

NULL signifies an unknown value or a value which doesn’t exist during the creation of the record. Field with NULL value is left blank during record creation.

It is not possible to test for NULL values using the comparison operators, such as =, <, or <>. A special keyword IS NULL and IS NOT NULL operators are used to test for NULL values instead of using comparison operators.

**SELECT** column\_list **FROM** **TABLE\_NAME**

**WHERE** column\_name **IS** **NULL**/**IS** **NOT** **NULL**;

When a NULL value is involved in a comparison operation, the result will be UNKNOWN.

 Which of the following keyword when used with a column name while creating the table signifies that it should always accept an explicit value of the given column data type?  
a) Is not null  
b) Null  
c) Not null  
d) Is null

Explanation: NOT NULL signifies that a column should always accept a value of the given data type. Is not null and Is null are used as comparison operators. They are not used while creating a table. NULL is used to assign an unknown value to a field.

The WHERE clause can be combined with NOT operator. The NOT operator displays a record if the condition(s) is NOT TRUE. It doesn’t display a record if the condition(s) is TRUE.

Select \* from Customers where not(country=’USA’ or country=’Germany’);

 The IN operator allows us to specify multiple values in a WHERE clause. It is used to replace when three or more AND and OR conditions are combined, to make the SQL query simple.

 Order By clause is used to sort the retrieved data in ascending or descending order. Group By clause is used to group the result-set by one or more columns. Order and Group are not valid SQL commands.

The ORDER BY keyword sorts the records in ascending order by default.

The columns specified in ORDER BY clause may or maynot be the columns selected in the SELECT column list.

Column Alias could be used in Order By clause. New name of the Attribute stated in the SELECT statement is to be used with ORDER BY clause to sort the result-set in ascending or descending order.

Instead of using column name in the ORDER BY clause, we can use the column position in the columns list of SELECT statement. Both of them yields the same result.

SELECT FirstName, LastName, Age FROM Employees ORDER BY LastName; -- Sorting by column name

SELECT FirstName, LastName, Age FROM Employees ORDER BY 2; -- Sorting by the second column in the SELECT list.

**SELECT** \* **FROM** **TABLE\_NAME** **ORDER** **BY** column1, column2;

----

SELECT \*

FROM Employees

ORDER BY Department ASC, Salary DESC;

1.The results will first be sorted by Department in ascending order.

2.For employees within the same department, results will be sorted by Salary in descending order.

**INSERT** **INTO** **TABLE\_NAME** **VALUES** (value1, value2, value3 . . .);

To add n number of rows into a table, this statement should be written n times.

**INSERT** **INTO** dept **VALUES** (7, ‘xxx’, **NULL**);

-------------------------------------------------------------

Rows can be copied from one table to another table using INSERT INTO statement.  
a) True  
b) False

Answer: a  
Explanation: We can use the INSERT INTO statement to add rows to a table where the values are derived from existing tables. In place of the VALUES clause, we use a subquery. Number of columns in the INSERT INTO clause should match with those in the subquery.

SELECT statement is used with INSERT INTO statement to copy rows from one table and insert them into another table. SELECT statement is used to select data from a table and INSERT INTO statement to insert in a different table.

INSERT INTO target\_table (column1, column2, ...)

SELECT column1, column2, ...

FROM source\_table

WHERE condition;

INSERT INTO EmployeesBackup (EmployeeID, FirstName, LastName, Salary)

SELECT EmployeeID, FirstName, LastName, Salary

FROM Employees

WHERE Salary > 50000;

--------------------------------------------------------------------------------

When the INSERT INTO statement is used with a SELECT statement, the VALUES clause is omitted because the data being inserted comes directly from the SELECT query rather than being explicitly provided.

INSERT INTO target\_table (column1, column2) SELECT column1, column2 FROM source\_table;

This syntax is used when INSERT INTO is used with SELECT statement. This statement is used to copy rows from one table and insert them into another table.

Using SQL, how can we copy all the data of a table and insert into in a different table if both the tables has same attributes?

INSERT INTO first SELECT \* FROM second;

CREATE TABLE second ( id INT, name VARCHAR(50), age INT );

CREATE TABLE first ( id INT, name VARCHAR(50), age INT );

INSERT INTO first SELECT \* FROM second;

To ensure no duplicate rows, you might need additional conditions like WHERE or use a DISTINCT clause in the SELECT statement:

INSERT INTO first

SELECT DISTINCT \*

FROM second;

UPDATE Employees

SET Salary = 50000,

Department = 'HR',

Status = 'Active'

WHERE EmployeeID = 101;

* Can update multiple set

Which of the following clause is used with SQL UPDATE statement when updating a table with data from another table?

When updating a table with data from another table, the **FROM** clause is used to specify the source table. This is often done in conjunction with a JOIN to match rows between the target and source tables.

**Given Tables:**

**Employees (Target Table):**

| **EmployeeID** | **Name** | **Salary** |
| --- | --- | --- |
| 1 | John | 50000 |
| 2 | Alice | 60000 |
| 3 | Bob | 70000 |

**SalaryUpdates (Source Table):**

| **EmployeeID** | **NewSalary** |
| --- | --- |
| 1 | 55000 |
| 3 | 75000 |

**Updating Employees Table:**

UPDATE Employees

SET Employees.Salary = SalaryUpdates.NewSalary

FROM SalaryUpdates

WHERE Employees.EmployeeID = SalaryUpdates.EmployeeID;

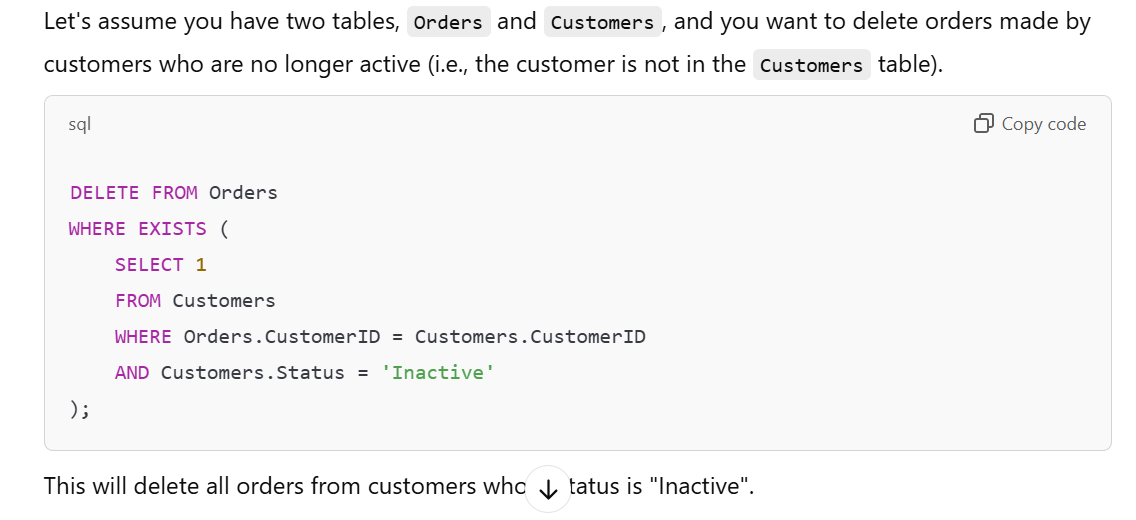
**Result:**

**Employees (After Update):**

| **EmployeeID** | **Name** | **Salary** |
| --- | --- | --- |
| 1 | John | 55000 |
| 2 | Alice | 60000 |
| 3 | Bob | 75000 |

It is possible to delete all rows in a table without deleting the table by using DELETE statement. Table structure, attributes, and indexes of a table will remain even after deleting all the records using DELETE statement.

Only one table can be deleted at a time using one DELETE statement. To delete n number of tables, delete statement should be written n number of times.



**SELECT 1**: The subquery doesn’t need to return specific data—just a true/false result indicating whether the condition holds.

It is possible to have one or more columns in addition to the SUM or AVG function in the SELECT statement. In those cases, these columns need to be part of the GROUP BY clause as well.

SELECT department, SUM(sales) FROM sales\_data GROUP BY department;

The SUM function can be used to calculate the total for an expression

SELECT SUM(revenue - cost) AS total\_profit FROM sales\_data;

The **SUM()** function is designed to work with **specific columns** or **arithmetic expressions** involving columns, but not with \* or ALL, because the operation involves adding up values from a specific numeric column or expression. It doesn't apply to all columns or entire records.

AVG and SUM functions can be used with select, Group by, having and distinct clauses. Order By cannot be used with these functions.

COUNT(1) has the same effect as COUNT(\*).

**COUNT(\*)** and **COUNT(1)** both count all rows in the result set, regardless of whether any column values are NULL.

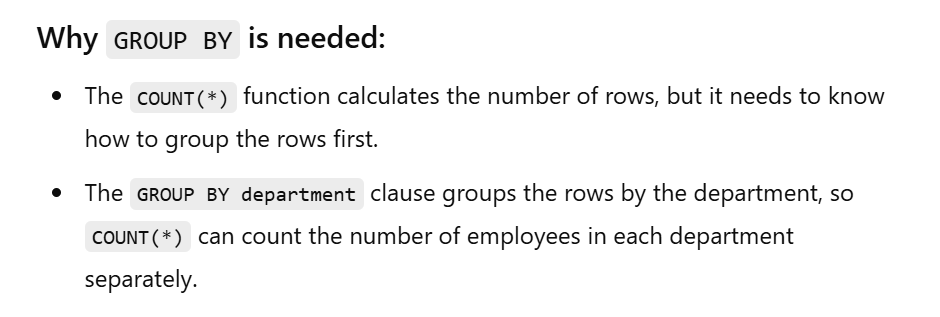
To count non-NULL values in a specific column, use **COUNT(column\_name)**.

 It is possible to have one or more columns in addition to the COUNT function in the SELECT statement. But these columns need to be a part of GROUP BY clause also.

SELECT department, COUNT(\*) AS num\_employees

FROM employees

GROUP BY department;



COUNT( ) function when combined with DISTINCT calculates the number of unique values. HAVING and GROUP BY clauses can also be used with COUNT function depending on the conditions, whereas order by cannot use any function or clause other than ASC and DESC.

It is acceptable to use spaces in an alias name. If the alias name contains spaces, we must enclose the alias name in double quotes or square brackets. This alias name is only valid within the scope of the SQL statement.

SELECT first\_name AS "Employee Name", last\_name AS "Employee Surname"

FROM employees;

SELECT first\_name AS [Employee Name], last\_name AS [Employee Surname]

FROM employees;

IN operator is used to remove the need of multiple OR condition in SELECT, INSERT, UPDATE or DELETE. It can be used as a shorthand for multiple OR operators.

The IN operator allows us to specify multiple values in a WHERE clause. Using this operator we can easily test if the expression matches any value in the list of values.

Any type of arguments, i.e. character, numeric, date etc. types can be used with IN operator. Character fields must be enclosed in single quotes.

Which of the following statement(s) can be used in place of list of values in an IN operator?  
a) Insert  
b) Update  
c) Select  
d) Modif

the use of a **SELECT** statement inside the **IN** operator is a **subquery**.

SELECT \*

FROM employees

WHERE department\_id IN (SELECT department\_id FROM departments WHERE location = 'New York');

A select statement can be substituted in place of values list in an IN operator. Values retrieved by that select statement is used as values list of IN operator. Value retrieved from select statement should match with the field mentioned in where clause.

 Other than AND, no operator can be used with BETWEEN.

 It is always used in WHERE clause. It cannot be used with any other clauses in SQL.

NOT BETWEEN operator to exclude a specific range of values from a database table.

The **BETWEEN** operator works as **value BETWEEN lower\_bound AND upper\_bound**.

In the case where you specify **max\_value** first and **min\_value** second, the query will return **no records**.

Using date fields with BETWEEN is similar to those of numeric and text field values.

Wildcard operators are used as an alternate for one or more than one characters while searching. A wildcard operator is used to substitute other characters in a string.

The **%** wildcard matches zero or more characters, and the **\_** wildcard matches exactly one character.

SELECT \* FROM customers

WHERE customer\_name LIKE 'A%';

This query will return all customers whose name starts with the letter "A".

SELECT \* FROM employees

WHERE employee\_id LIKE '\_\_\_2';

This query will return all employees whose ID ends with the digit "2" and has exactly three characters.

MS Access uses a question mark (?) instead of underscore (\_). MS Access and SQL server can use [char list] and [!char list] wildcards.

 [char list] defines sets and ranges of characters to match. It is used to fetch matching set or range of characters specified inside the brackets.

**[!m-r]\_\_\_%** and **[^m-r]\_\_\_%** are the **same** in terms of behavior.

SELECT \* FROM users

WHERE username LIKE '%@gmail.com';

This query would find all usernames that end with @gmail.com, including the special @ character.

The pattern ‘x\_%\_%’ matches which of the following strings? a) Strings which starts with x and contain any number of characters b) Strings which have x in any position c) Strings which start with ‘a’ and contain at least 2 more characters d) Strings which start with ‘a’ and contain exactly 2 characters

Answer: c  
Explanation: % represents zero or more characters, whereas \_ represents exactly one character. As the given pattern has two % characters, the actual string may contain zero characters in that place, whereas in the positions with \_ , there must be a character. So, the given pattern should contain at least 2 more characters along with a.

All the non-aggregate function columns of the SELECT statement must be listed in the GROUP BY clause. Otherwise, the query displays an error.

The WHERE clause cannot be used to restrict groups. It is used to retrieve rows based on a certain condition, but it cannot be applied to grouped result. The HAVING clause is used to filter the result set of GROUP BY clause.

 The HAVING clause is used with the GROUP BY clause to specify a filter condition for a group or an aggregate. HAVING clause places conditions on groups created by the GROUP BY clause.

A SELECT query can contain both a WHERE and a HAVING clause, but in that case WHERE must appear before GROUP BY and HAVING must appear after GROUP BY.

In the absence of GROUP BY clause, the HAVING clause works like a WHERE clause. The difference between the HAVING and the WHERE clause is that the WHERE clause is used to filter rows, while the HAVING clause is used to filter groups of rows.

SELECT Product, SUM(Sales)

FROM sales WHERE Region = 'North'

GROUP BY Product;

WHERE clause filters the rows where Region is 'North' **before** any aggregation happens.

SELECT Product, SUM(Sales)

FROM sales

GROUP BY Product

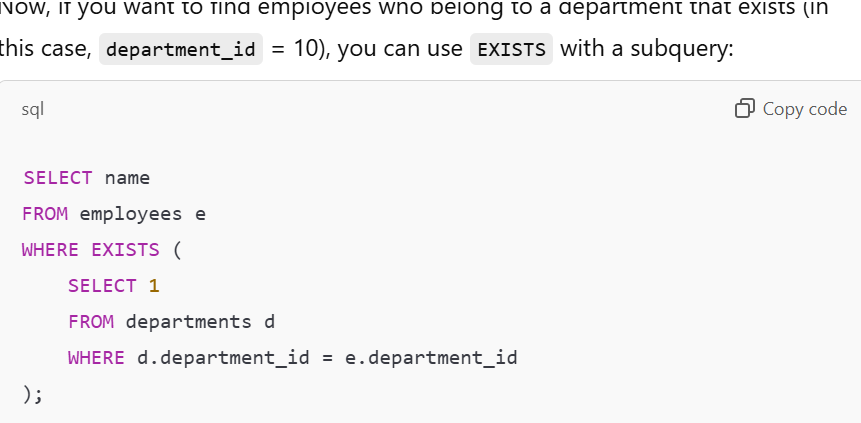
HAVING SUM(Sales) > 30;

The HAVING clause filters the results **after** the aggregation is done (i.e., after summing up the Sales for each Product).

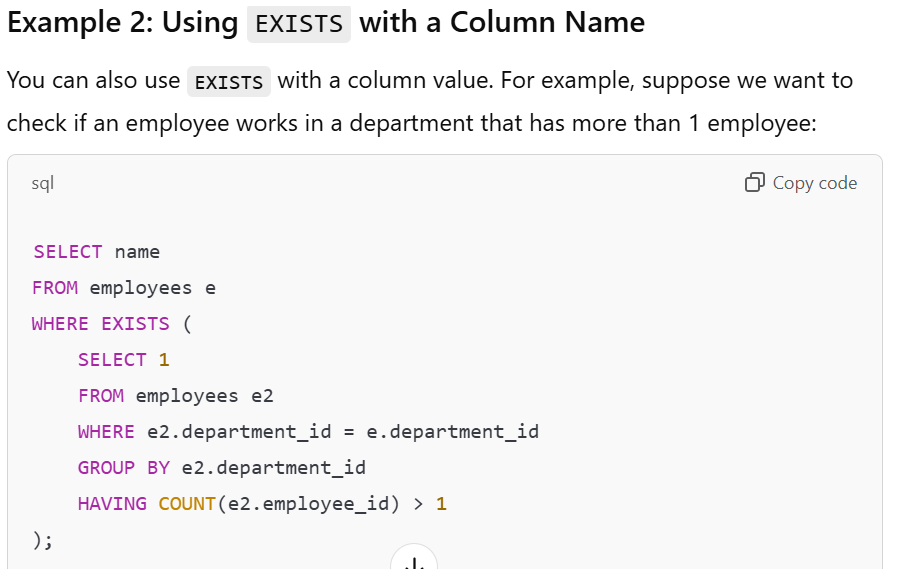
Use **WHERE** for filtering individual rows based on column values.Use **HAVING** for filtering groups formed by GROUP BY or based on aggregate function results.

The EXISTS condition in SQL is used to check whether the result of a correlated nested query is empty or not. It is used to test for the existence of a record in a subquery.

The result of EXISTS is a Boolean value. It returns true if the subquery returns one or more records and false if the subquery returns no records.



This query checks if a department\_id exists in the departments table for each employee. The SELECT 1 inside the subquery is just a placeholder because EXISTS only cares about the existence of rows, not the specific data returned.



**Explanation**: This query checks if there exists at least one department with more than one employee by using EXISTS with a subquery that groups by department\_id and counts employees.

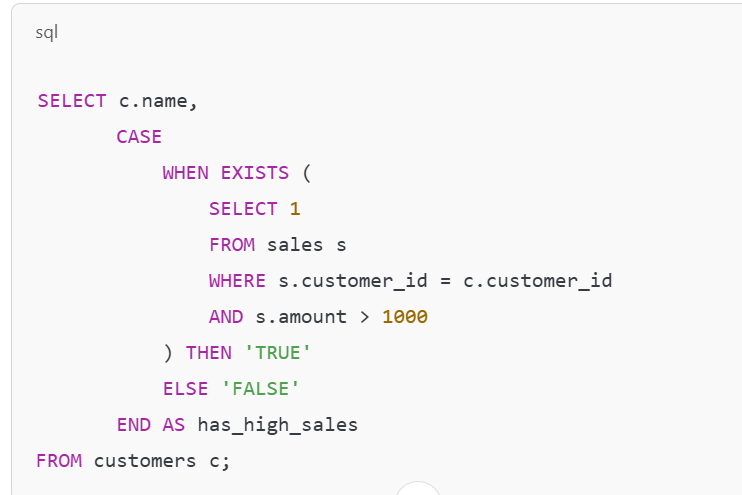
It returns employees from departments with more than one employee.



**Explanation**: This query checks if there exists at least one employee in the same department whose salary is greater than the average salary for that department.

Select column\_list  
From table\_name  
Where EXISTS (subquery)

When EXISTS operator is used, the process of inner query execution repeats as many times as there are outer query rows. If there are ten rows that can result from main query, the subquery will be executed ten times.



**Output for the Above Query:**

| **name** | **has\_high\_sales** |
| --- | --- |
| Alice | FALSE |
| Bob | FALSE |
| Charlie | FALSE |
| Dave | FALSE |

This explicitly shows whether each customer satisfies the EXISTS condition.

If id from the outer query does not match any id in the subquery, the EXISTS condition evaluates to FALSE, and that row is excluded from the result.

The query will return an **empty result set** in such cases.

When EXISTS operator is used, the process of inner query execution repeats as many times as there are outer query rows. If there are ten rows that can result from main query, the subquery will be executed ten times.

EXISTS operator cannot be prefixed using In, Like or between operators. NOT operator can be used as a prefix for EXISTS operator.

NOT EXISTS operator returns true if no records are returned by the subquery and false if at least one record is returned by the sub query.

The ANY and ALL operators are used with a WHERE or HAVING clause in SQL. ALL can be used with SELECT clause whereas ANY cannot be used with it. ANY and ALL cannot be used with GROUP BY.

SELECT name

FROM employees

WHERE salary > ANY (SELECT salary FROM employees WHERE department = 'HR');

This query selects employees whose salary is greater than **at least one** salary in the HR department.

1. **Outer Query Execution**:
   * The outer query evaluates each employee's salary:
     + **Bob (2500)**: Not greater than 3000, 5000, or 2000 → Not included.
     + **Dave (4000)**: Greater than 3000 and 2000 → Included.
     + **Alice, Charlie, Eve**: Not part of the outer query's filtering.

The ANY operator evaluates if the condition matches **any one value** in the list or subquery. In this case, only Dave's salary satisfies the condition.

--

SELECT name FROM employees WHERE salary > ALL (SELECT salary FROM employees WHERE department = 'HR');

This query selects employees whose salary is greater than **all salaries** in the HR department.

**3. Outer Query Execution:**

* The outer query evaluates each employee's salary:
  + **Alice (3000)**: Not greater than 5000 → Not included.
  + **Bob (2500)**: Not greater than 5000 → Not included.
  + **Charlie (5000)**: Not greater than 5000 → Not included.
  + **Dave (4000)**: Not greater than 5000 → Not included.
  + **Eve (2000)**: Not greater than 5000 → Not included.

For the condition to be TRUE, the employee's salary must exceed the **maximum value** from the subquery, which is 5000.

ANY operator compares a value with the values in a list or results of a subquery and evaluates to true if the result of sub query contains at least one row.

ALL operator is used to compare a value to every value in another value set or result from a subquery. It returns true if all of the subquery values meet the condition.

The ALL and ANY operators must be preceded by comparison operators like =, < >, !=, >, >=, <, <=.

If a subquery of ALL operator returns zero rows, the condition evaluates to TRUE. If the subquery returns zero rows, it means the whole expression x > ALL (zero rows) evaluates to TRUE.

SELECT name

FROM employees

WHERE salary > ALL (SELECT salary FROM employees WHERE department = 'Engineering');

**employees**:

| **employee\_id** | **name** | **salary** | **department** |
| --- | --- | --- | --- |
| 1 | Alice | 3000 | HR |
| 2 | Bob | 4000 | Sales |
| 3 | Charlie | 5000 | HR |
| 4 | Dave | 6000 | Marketing |

**Explanation:**

1. **Subquery Execution**:
   * Subquery: SELECT salary FROM employees WHERE department = 'Engineering'.
   * Result: **Zero rows**, because there are no employees in the Engineering department.
2. **Condition Evaluation**:
   * The ALL operator checks: salary > ALL (zero rows).
   * Since the subquery returns zero rows, the condition evaluates to TRUE for all rows in the outer query.

 If subquery of a query with ANY operator returns zero rows, the condition evaluates to FALSE. If the subquery returns zero rows, it means the whole expression x > ANY (zero rows) evaluates to FALSE.

SELECT name

FROM employees

WHERE salary > ANY (SELECT salary FROM employees WHERE department = 'Engineering');

**Table:**

**employees**:

| **employee\_id** | **name** | **salary** | **department** |
| --- | --- | --- | --- |
| 1 | Alice | 3000 | HR |
| 2 | Bob | 4000 | Sales |
| 3 | Charlie | 5000 | HR |
| 4 | Dave | 6000 | Marketing |

**Explanation:**

1. **Subquery Execution**:
   * Subquery: SELECT salary FROM employees WHERE department = 'Engineering'.
   * Result: **Zero rows**, because there are no employees in the Engineering department.
2. **Condition Evaluation**:
   * The ANY operator checks: salary > ANY (zero rows).
   * Since the subquery returns zero rows, the condition evaluates to FALSE for all rows in the outer query.
3. **Result**:
   * The query returns **no rows** because the ANY condition evaluates to FALSE.

When the subquery of a query with ANY returns **zero rows**, the condition evaluates to **FALSE** because there are no values to satisfy the condition.

----------------------------------

 Minimum required condition for joining table, is N-1 where N is the number of tables.

 If on keyword is not used in a query, then the join condition must be specified in the where clause.

SELECT e.name, d.department\_name

FROM employees e, departments d

WHERE e.department\_id = d.department\_id;

LEFT JOIN returns all records from the left table, and the matched records from the right table. We can use LEFT OUTER JOIN instead of LEFT JOIN; both of them produce same result.

RIGHT JOIN returns all records from the right table, and the matched records from the left table. We can use RIGHT OUTER JOIN instead of RIGHT JOIN; both of them produce same result.

**Summary of Intersection (Set Operations) for Each Join:**

* **INNER JOIN**: Table1 ∩ Table2 (Intersection of both sets)
* **LEFT JOIN**: Table1 U Table2 (Union, with NULL for non-matching rows in Table2)
* **RIGHT JOIN**: Table1 U Table2 (Union, with NULL for non-matching rows in Table1)
* **FULL OUTER JOIN**: Table1 U Table2 (Union, with NULL for non-matching rows in both tables)

SELF JOIN is used to join or compare a table with itself. SQL self joins are used to compare values of a column with values of same or another column in the same table

**Key Points:**

* **Self Join**: Joins a table to itself.
* **Alias**: We use aliases to differentiate the two instances of the same table in the query.

**Example:**

Let’s say we have an **employees** table that looks like this:

| **employee\_id** | **name** | **manager\_id** |
| --- | --- | --- |
| 1 | Alice | NULL |
| 2 | Bob | 1 |
| 3 | Charlie | 1 |
| 4 | David | 2 |
| 5 | Eve | 2 |

In this table:

* employee\_id is the unique ID for each employee.
* manager\_id represents the employee\_id of the employee's manager. If the employee has no manager, this value is NULL.

**Scenario:**

We want to find out the **name of each employee along with the name of their manager**.

**SQL Query with SELF JOIN:**

SELECT e.name AS Employee, m.name AS Manager

FROM employees e

LEFT JOIN employees m

ON e.manager\_id = m.employee\_id;

**Explanation:**

* We are **joining** the employees table with itself.
* e is the alias for the first instance of the table (representing employees).
* m is the alias for the second instance of the table (representing managers).
* We use LEFT JOIN to ensure that even employees who do not have a manager (i.e., their manager\_id is NULL) are included in the result.
* The ON clause compares the manager\_id of the employee (e.manager\_id) with the employee\_id of the manager (m.employee\_id).

**Result:**

| **Employee** | **Manager** |
| --- | --- |
| Alice | NULL |
| Bob | Alice |
| Charlie | Alice |
| David | Bob |
| Eve | Bob |

**Explanation of Result:**

* **Alice** has no manager, so the manager field is NULL.
* **Bob** and **Charlie** report to **Alice**, so their manager is listed as **Alice**.
* **David** and **Eve** report to **Bob**, so their manager is listed as **Bob**.

**When to Use SELF JOIN:**

* When comparing rows within the same table, like hierarchical structures (e.g., employee-manager relationships).
* When you need to compare data from different rows of the same table based on some condition (e.g., checking if a product's price is greater than another product's price).

**Conclusion:**

A **SELF JOIN** is a powerful tool in SQL when dealing with hierarchical or relational data within the same table.

There is no keyword as SELF JOIN in SQL. But it is referred when a table joins with itself.

We must use a table alias to distinguish the left table from the right table of the same table in a single query when we perform a SELF JOIN.

**Example:**

Consider the following **employees** table:

| **employee\_id** | **name** | **manager\_id** |
| --- | --- | --- |
| 1 | Alice | NULL |
| 2 | Bob | 1 |
| 3 | Charlie | 1 |
| 4 | David | 2 |
| 5 | Eve | 2 |

We want to get the **employee's name** along with their **manager's name**. To do this, we will use a **self-join** on the employees table.

**SQL Query (without SELF JOIN keyword):**

SELECT T.name AS Employee, S.name AS Manager

FROM employees T

JOIN employees S ON T.manager\_id = S.employee\_id;

**Result:**

| **Employee** | **Manager** |
| --- | --- |
| Bob | Alice |
| Charlie | Alice |
| David | Bob |
| Eve | Bob |

**SELF JOIN** is simply the process of joining a table with itself by using **aliases**.

There is no SQL keyword for "SELF JOIN"; it is just a **concept** used when a table is joined to itself.

The UNION operator is used to combine the result-set of two or more SELECT statements. By default it selects only distinct values.

The number of the columns must be the same in all SELECT statements whose result-sets are to be combined using UNION operator.

The order of the columns must be the same in all SELECT statements with UNION operator between them.

The columns listed in the SELECT statements with UNION operator must have similar data types. Those columns can also have date data type.

A set of SELECT statements are said to be union-compatible only when the fields used in those statements are in same order, same number and same data type.

The UNION operation eliminates the duplicate rows from the combined result set, by default. To fetch the duplicate values too UNION ALL must be used in the place of UNION.

Column names in the SELECT statements with UNION operator can be different but their data types must be same.

SELECT id, name FROM employees

The column names in the result-set are usually the column names in the first SELECT statement of the UNION.

UNION

SELECT employee\_id, employee\_name FROM contractors;

Assuming the following sample data:

* **employees table**:

| **id** | **name** |
| --- | --- |
| 1 | Alice |
| 2 | Bob |

* **contractors table**:

| **employee\_id** | **employee\_name** |
| --- | --- |
| 3 | Charlie |
| 4 | David |

The result of the above UNION query will be:

| **id** | **name** |
| --- | --- |
| 1 | Alice |
| 2 | Bob |
| 3 | Charlie |
| 4 | David |

**Conclusion:**

The column names in the result set of a UNION operation are derived from the first SELECT statement, and they are used for all the rows returned from the combined queries.

**Key Differences Between UNION and JOIN in SQL:**

| **Aspect** | **UNION** | **JOIN** |
| --- | --- | --- |
| **Purpose** | Combines the result sets of two or more SELECT queries into one result set. | Combines columns from two or more tables based on a related column. |
| **Data Structure** | **Rows**: Combines rows from multiple result sets. | **Columns**: Combines columns from different tables. |
| **Number of Tables** | Involves multiple SELECT queries, each from one table (or more). | Involves multiple tables in a single query. |
| **Result Type** | A single result set with rows stacked vertically. | A single result set with columns from multiple tables side by side. |
| **Duplicates** | By default, removes duplicate rows. Can use UNION ALL to keep duplicates. | Keeps all rows, including duplicates, unless DISTINCT is used. |
| **Column Matching** | Requires the same number of columns with compatible data types in each SELECT statement. | Combines columns from different tables based on a related condition (e.g., a foreign key). |
| **Condition** | No condition needed for merging rows (but can be used to filter in each query). | Requires a condition (e.g., ON clause) to specify how to match rows between tables. |
| **Use Case** | When you need to append results from two or more queries. | When you need to merge data from two or more related tables. |

**UNION** is used when you want to combine the result sets of two or more queries vertically (adding more rows), with the same number of columns.

**JOIN** is used when you want to combine data from multiple tables horizontally (adding more columns), based on a common condition.

The databases present in the SQL server can be listed using the statement – Select name from master.sys.databases order by name. Here, master.sys.databases means that the owner of the databases is calling all the databases in his system and order by clause is used to order the names according to the specified condition.

 IF NOT EXISTS command helps us instruct the SQL server to check the existence of a database with a similar name before creating a database. It is helpful if you are working on a server that can be accessed by multiple people because there is a probability that there might be a database with a similar name that has been created by another person.

IF NOT EXISTS (SELECT \* FROM sys.databases WHERE name = 'YourDatabaseName')

BEGIN

CREATE DATABASE YourDatabaseName;

END;

The “AS COPY OF” command is used to create a database as a copy of another source database. It can be used as follows –

**CREATE** **DATABASE** database\_name **AS** COPY **OF** source\_database\_name

 The DROP command is used to delete a database in SQL. The same command can be used to delete a table within the database. The DELETE command is used to delete the specific information present inside the table. The TRUNCATE command is used to delete complete data from the table. Be careful while using the DROP command for deleting the database as it will affect all the tables present inside the database.

A database can be dropped regardless of its state i.e. offline, read-only, suspect, etc. To view the current state of a database we can use the sys.databases catalog view.

In SQL, a dropped database can only be re-created by restoring a backup of it. But the database snapshots cannot be backed up and, therefore, cannot be restored.

**Dropped databases** require backups to be restored.

**Database snapshots** are not backed up and cannot be restored independently.

 You cannot drop a database currently being used i.e. open for reading or writing by any user. One way to remove users from the database is to use the ALTER DATABASE command to set the database to SINGLE\_USER.

In SQL Server, you cannot drop a database that is currently in use—i.e., if it is open for reading or writing by any user. One way to ensure the database is no longer in use is to set it to **SINGLE\_USER** mode, which allows only one connection to the database. This effectively removes other users or connections.

You can use the following command to set the database to **SINGLE\_USER** mode:

ALTER DATABASE YourDatabaseName

SET SINGLE\_USER

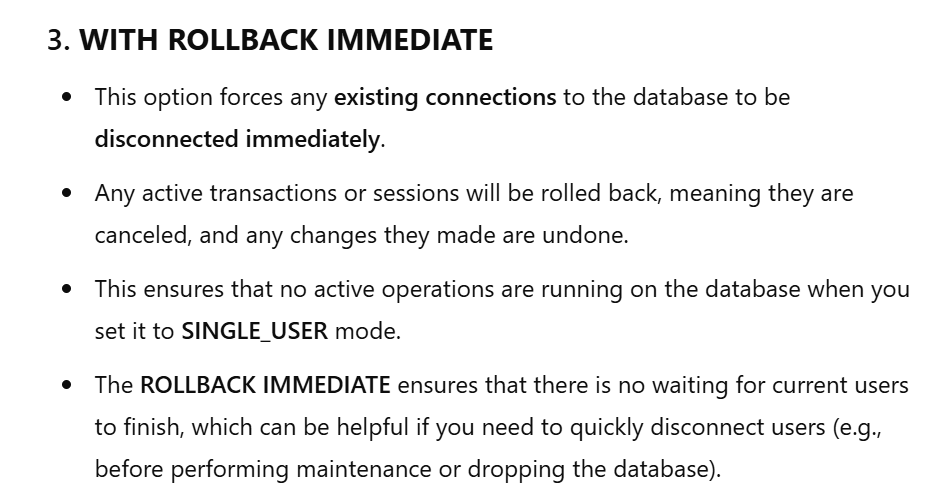
WITH ROLLBACK IMMEDIATE;

* WITH ROLLBACK IMMEDIATE ensures that any existing transactions or connections are immediately rolled back and disconnected.
* After the database is in **SINGLE\_USER** mode, you can safely drop the database.

To drop the database after setting it to **SINGLE\_USER** mode:

DROP DATABASE YourDatabaseName;

This process ensures that no users are actively using the database when attempting to drop it.



The IF EXISTS command is used with the DROP command to conditionally drop a database only if it already exists in the SQL Server. If we try to drop a database that is not present, SQL will give us an error. To prevent an error from occurring we can use the IF EXISTS option.

IF EXISTS (SELECT \* FROM sys.databases WHERE name = 'YourDatabaseName')

BEGIN

DROP DATABASE YourDatabaseName;

END;

Or

DROP DATABASE IF EXISTS YourDatabaseName;

CANDIDATE KEY is not present as a constraint in SQL. A Candidate key is the set of columns that can become the Primary Key. The column having the UNIQUE constraint cannot have any duplicate value. The column having the NOT NULL constraint cannot have any null value. The PRIMARY KEY constraint is a combination of UNIQUE and NOT NULL.

A **Candidate Key** is a potential primary key, consisting of one or more columns, which can uniquely identify rows.

The **UNIQUE** constraint allows only distinct non-null values but can allow multiple NULL values (depending on the database system).

The **PRIMARY KEY** constraint combines **UNIQUE** and **NOT NULL**, ensuring that the selected candidate key uniquely identifies rows without allowing nulls.

Which of the following SQL statement is used to create a table by copying only the selected columns from another table?

**CREATE** **TABLE** new\_tablename **AS** (

**SELECT** Column1, Column2...Columnn

**FROM** old\_tablename);

**CREATE** **TABLE** new\_tablename **AS** (

**SELECT** \* **FROM** old\_tablename **WHERE** condition);

is used to create a table by copying all the columns from another table.

**Implicit Join:**

**CREATE** **TABLE** new\_tablename **AS** (

**SELECT** Column1, Column2...ColumnN

**FROM** old\_tablename1, old\_tablename2...old\_tablenamen);

is used to create a table by copying selected columns from multiple tables.

**Explicit Join (JOIN statement)**:

CREATE TABLE new\_tablename AS

SELECT e.emp\_name, d.dept\_name

FROM employees e

JOIN departments d

ON e.dept\_id = d.dept\_id;

**Key Differences:**

* **Implicit Join (Comma-Separated Tables)**: Produces a Cartesian product (all combinations of rows), which is rarely useful unless you intentionally want every combination of rows.
* **Explicit Join (JOIN statement)**: Allows for meaningful connections between tables by specifying conditions (like ON), making it much more useful and efficient for most use cases.

In summary, **always prefer using JOIN** when working with multiple tables, as it gives you control over how tables are related and produces more relevant results.

The DROP command is used to delete the structure of a table in SQL. The same command can be used to delete a database. The DELETE command is used to delete the specific information present inside the table. The TRUNCATE command is used to delete complete data from the table.

 DROP TABLE IF EXISTS table\_name

if a table is referenced by a **FOREIGN KEY** constraint in another table, you cannot drop that table directly using the DROP command. Doing so will result in an error because dropping the table would violate referential integrity rules.

**Tables with Foreign Key:**

**Table 1: departments**

| **dept\_id** | **dept\_name** |
| --- | --- |
| 101 | HR |
| 102 | IT |

**Table 2: employees**

| **emp\_id** | **emp\_name** | **dept\_id** |
| --- | --- | --- |
| 1 | Alice | 101 |
| 2 | Bob | 102 |

In this case, employees.dept\_id references departments.dept\_id.

**Attempt to Drop departments:**

DROP TABLE departments;

**Result**: Error, because departments is referenced by the foreign key in employees.

**Solution:**

1. Drop the foreign key constraint:

ALTER TABLE employees

DROP CONSTRAINT fk\_dept\_id; -- Replace `fk\_dept\_id` with the actual constraint name.

1. Drop the departments table:

DROP TABLE departments;

**Summary:**

The **FOREIGN KEY** constraint ensures referential integrity. You must either remove the constraint or drop the referencing table before you can drop the referenced table.

Which of the following statement is used to drop a table present in another database?  
DROP TABLE database\_name.dbo.table\_name

 In SQL, the ALTER command cannot be used to perform the restore column function. The ALTER command is used to add, drop or modify an existing column in a table.

**ALTER** **TABLE** **TABLE\_NAME**

**MODIFY** **COLUMN** column\_name datatype;

Rename the table:

**ALTER** **TABLE** **TABLE\_NAME**

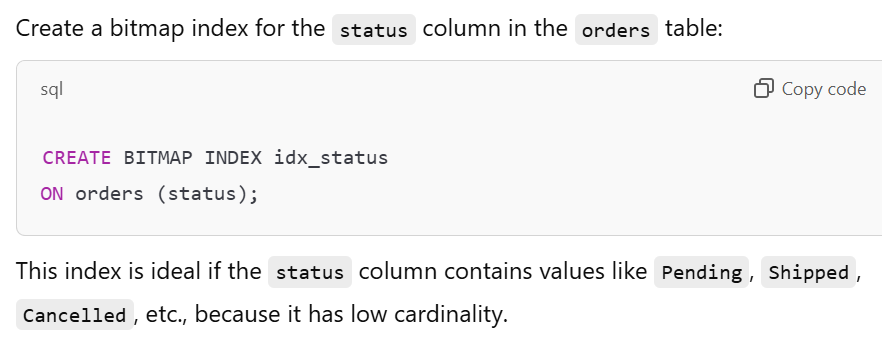
**RENAME** **TO** new\_table\_name;

Rename the column:

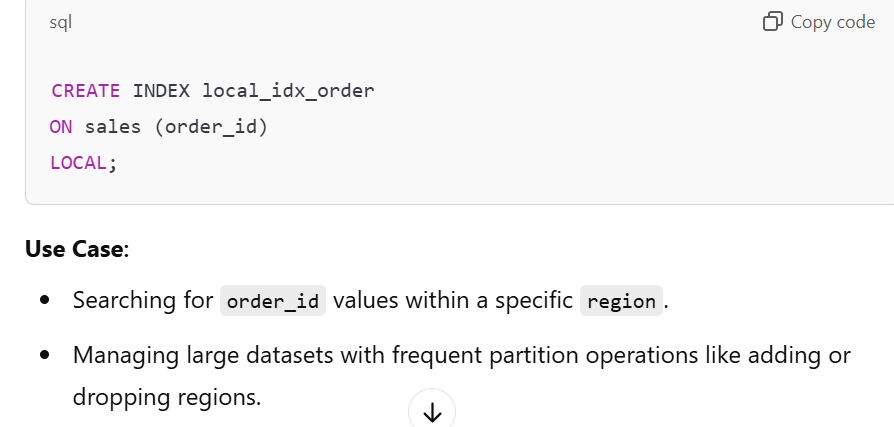
**ALTER** **TABLE** **TABLE\_NAME**

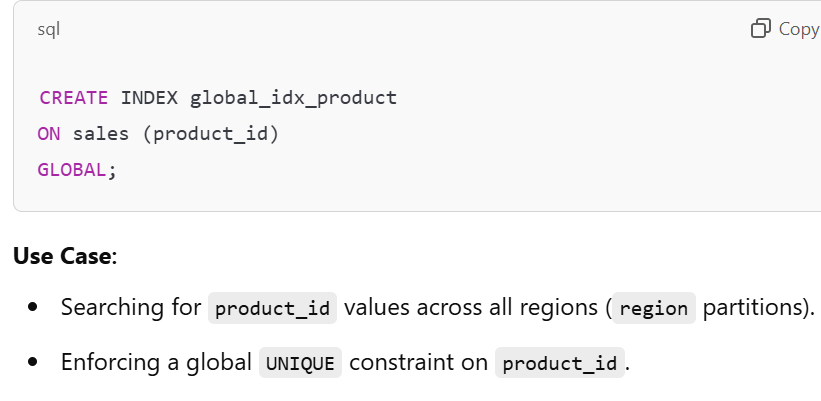
**RENAME** **COLUMN** old\_name **TO** new\_name;

**CREATE** BITMAP **INDEX** index\_name **ON** **TABLE\_NAME** (column\_name);



 In SQL, the Global and Local indexes are used when there is a partition in a table. Global index has one to many relationships which allows index partition to map to many table partitions. Local indexes are indexes where there is one to one mapping between index partition and table partition.





**Comparison: Global vs. Local Indexes**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Global Index** | **Local Index** |
| **Relationship** | One-to-many | One-to-one |
| **Scope** | Spans all table partitions | Aligned with each table partition |
| **Partition Independence** | Not independent; changes affect all partitions | Independent; changes only affect corresponding partition |
| **Maintenance** | Complex; requires rebuilding during partition operations | Simpler; localized to specific partitions |
| **Global Constraints** | Supports global constraints like UNIQUE | Does not support global constraints |
| **Query Efficiency** | Efficient for global queries across partitions | Efficient for partition-specific queries |
| **Storage** | Requires additional space to manage globally | Requires space per partition |

CREATE TABLE users (

id INT IDENTITY(1,1) PRIMARY KEY, -- Starts at 1, increments by 1 -- Auto-incrementing column

name NVARCHAR(50)

);

CREATE SEQUENCE user\_id\_seq

START WITH 1

INCREMENT BY 1;

-- Using the sequence in a table:

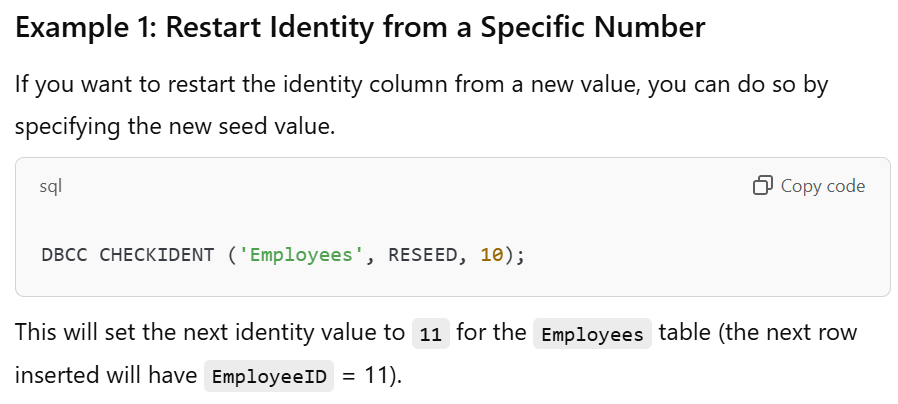
INSERT INTO users (id, name) VALUES (NEXT VALUE FOR user\_id\_seq, 'Alice');

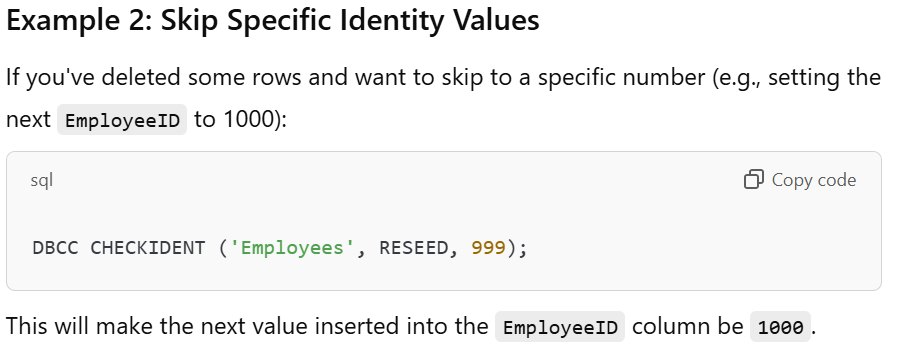
**Comparison: AUTO INCREMENT vs. SEQUENCES**

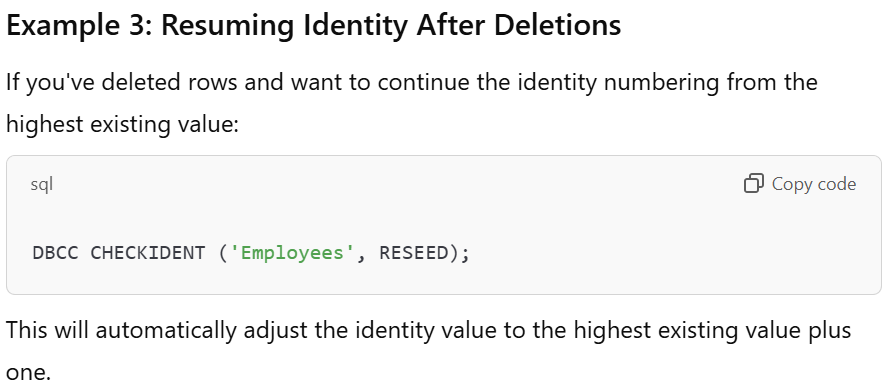
|  |  |  |
| --- | --- | --- |
| **Feature** | **AUTO INCREMENT** | **SEQUENCES** |
| **Tight Coupling** | Tied to a specific table column | Independent of any table |
| **Flexibility** | Limited to one table | Can be shared across tables |
| **Control** | Basic (e.g., increment by 1) | Full control (start, increment, cycle, etc.) |
| **Gaps** | Gaps when rows are deleted | Can be explicitly managed |
| **Complexity** | Easy to use | Requires explicit management |

IDENTITY cannot be applied to a column that already has data.

If you want to skip specific IDs or restart the numbering, you can use the DBCC CHECKIDENT command.







DBCC CHECKIDENT is a powerful command for controlling the identity column's behavior in SQL Server, especially when you want to restart or skip specific values.

**CREATE** **TABLE** **TABLE\_NAME** (

column1 datatype **CONSTRAINT** **AUTO\_INCREMENT**,

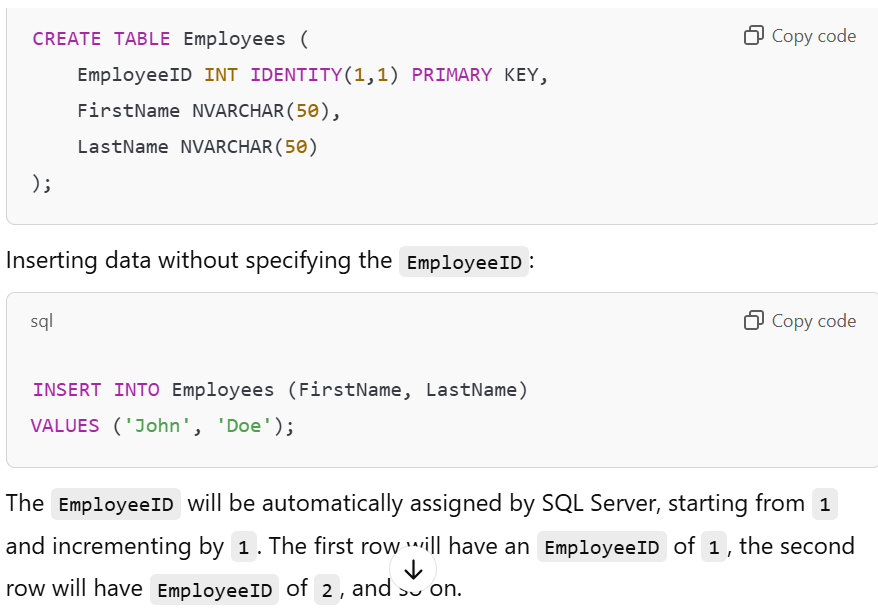
column2 datatype **CONSTRAINT**,

...

columnn datatype **CONSTRAINT**);

We can change the default value of the AUTO INCREMENT field by which a record is increased. By default, the value is 1 but we can change the value as according to our need.

ALTER TABLE table\_name AUTO\_INCREMENT=100;



column\_name datatype constraint AUTOINCREMENT (10,5);

column\_name datatype **CONSTRAINT** AUTOINCREMENT (10,5);

Here, the starting value for the column would be 10 and it would be automatically increased by 5 every time a new record is added to the column. Also, we use the AUTOINCREMENT keyword in SQL Server rather than the AUTO\_INCREMENT keyword which is used in MySQL.

DECIMAL (5,2)?  
c) 999.99

the CREATE TYPE command can be used to create new user defined datatypes. It can be used as –

**CREATE** **TYPE** Currency **AS** **DECIMAL** (8,4);

A TINYTEXT column has a maximum length of 255 characters.

Which of the following datatype is used for storing a maximum of one element from a predefined list consisting a maximum of 65535 elements?  
d) ENUM

What is the default size of n in a column that has the datatype VARCHAR(n)?  
c) 1

**COLUMN** Name: Id, Description: Nine-digit roll **NUMBER** starting **WITH** R, Example: R100218066

 VARCHAR2(10) will be the most suitable for the given column. VARCHAR is used because the ID is a combination of both numbers and letters. 10 specifies the maximum length of the ID that can be stored.

**COLUMN** Name: Student\_Photo, Description: Image **OF** every student

In SQL, the BLOB datatype is used to binary large object that can hold a variable amount of data. It can also be used to hold pictures. In mysql

CREATE TABLE Images (

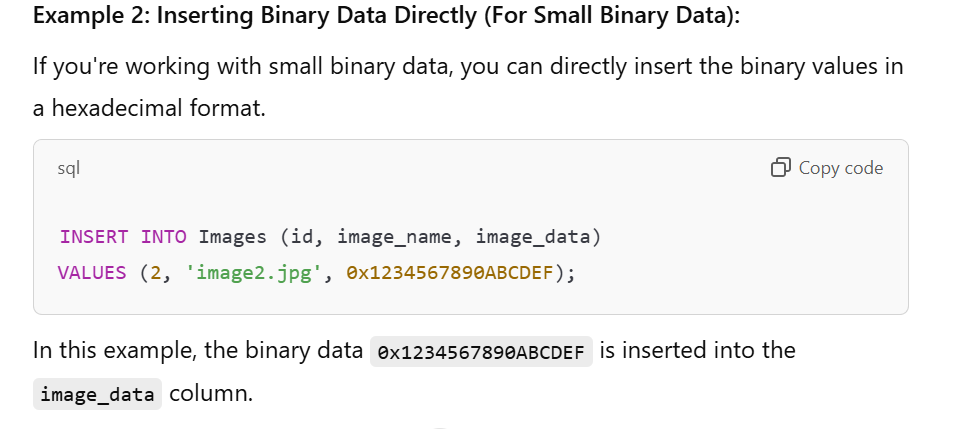
id INT PRIMARY KEY,

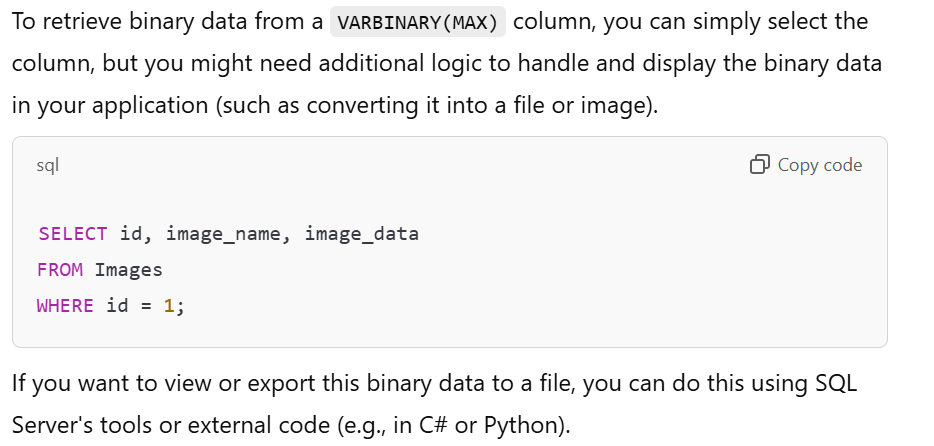
image\_name VARCHAR(255),

image\_data VARBINARY(MAX)

);







CURRENT\_DATE() function is a synonym of the CURDATE() function and can be used interchangeably.

CURRENT\_TIME() function is a synonym of the CURTIME() function and can be used interchangeably.

 The %c format string in the DATE\_FORMAT() function is used to specify the Month number format which is given by 0 for January, 1 for February and so on.

DAYOFWEEK() function is used to get the weekday index value (1 = Sunday, 2 = Monday, … , 7 = Saturday).

The NOW() function is used to return the current date and time in the YYYY-MM-DD HH:MM:SS

TIME\_TO\_SEC(’22:23:00’);

add 1 1:1:1.000002 to 1998-12-30 23:59:59.999998 we will get 1999-01-01 01:01:01.000000.

Let’s break this down step by step:

1. **Start time**: 1998-12-30 23:59:59.999998
2. **Addition**: 1 1:1:1.000002 (1 day, 1 hour, 1 minute, 1 second, and 2 microseconds)

**Step 1: Add 1 day** 1998-12-30 → 1998-12-31

Now, the time is 1998-12-31 23:59:59.999998.

**Step 2: Add 1 hour** 23:59:59.999998 → 00:59:59.999998 (roll over to 1999-01-01).

Now, the time is 1999-01-01 00:59:59.999998.

**Step 3: Add 1 minute** 00:59:59.999998 → 01:00:59.999998.

**Step 4: Add 1 second** 01:00:59.999998 → 01:01:00.999998.

**Step 5: Add 2 microseconds** 01:01:00.999998 → 01:01:01.000000.

**Final Result:**

1999-01-01 01:01:01.000000

We can perform insert and update in an updatable view only. Read-only views are not updatable.

**CREATE** **VIEW** view\_name **AS**

**SELECT** column1, column2..... columnn

**FROM** **TABLE\_NAME** **WHERE** condition;

The WITH CHECK option is a CREATE VIEW statement option. The purpose of the WITH CHECK option is to ensure that all UPDATE and INSERTs satisfy the conditions in the view definition.

There are 2 types of Views in SQL: Simple View and Complex View. Simple views constitute only of a single base table. Complex views can be constituted from more than one base table.

For the INSERT command to work on a view, all the NOT NULL columns of the underlying base table must be present in the view. This ensures that the mandatory columns in the base table receive values during the INSERT operation.

Here’s an example to illustrate this:

Q- in order for the INSERT command to function and enter rows in a view, all the NOT NULL columns from the base table must be included in the view while creating it

**Base Table**

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

FirstName VARCHAR(50) NOT NULL,

LastName VARCHAR(50) NOT NULL,

DepartmentID INT NULL,

Salary DECIMAL(10, 2) NULL

);

**View Creation**

If we create a view without including all NOT NULL columns, the INSERT operation will fail. Let’s create two views:

1. **View Missing a NOT NULL Column (Invalid for INSERT)**

CREATE VIEW EmployeeView1 AS

SELECT EmployeeID, LastName

FROM Employees;

In this view, the column FirstName is not included, but since it is marked as NOT NULL, this view **cannot** be used for an INSERT operation.

1. **View Including All NOT NULL Columns (Valid for INSERT)**

CREATE VIEW EmployeeView2 AS

SELECT EmployeeID, FirstName, LastName

FROM Employees;

This view includes both FirstName and LastName, which are NOT NULL. Hence, it is valid for INSERT.

--------

DELETE FROM View\_name

FROM TABLE\_NAME WHERE condition; --wrong

The DELETE operation on a view automatically maps to the underlying base table(s), so referencing TABLE\_NAME explicitly is unnecessary and invalid.

DELETE FROM View\_name

WHERE condition; ---correct

DELETE FROM EmployeeView

FROM Employees WHERE EmployeeID = 1;

This will throw a syntax error because the DELETE statement does not support an additional FROM clause referencing the base table.

When using the BETWEEN operator, the starting and the ending value is always included. The given values could be integers, texts or dates.

The EXISTS operator is used to search for the presence of a row in a specified table for a given condition. It returns true when value is found.

SELECT \*

FROM Customers

WHERE EXISTS (

SELECT 1

FROM Orders

WHERE Customers.CustomerID = Orders.CustomerID

);

SQL hosting is the process which used for the web server to have access to a database-system that uses the SQL language. This helps our website to store and retrieve data from the database.

When websites requires only a simple database then the MS Access can be used. It does not works well with a high traffic database driven websites. It is not as powerful as MySQL, MS SQL Server or Oracle database.

The MS SQL Sever uses Transact SQL or T-SQL ass its language. PL/SQL is used by Oracle database as its language.

The SQL Server database management server is owned by Microsoft and can only run on Windows operating system. Whereas, the database systems like Oracle, MySQL and MongoDB can work on Windows, Mac as well as Linux Operating Systems.

**SQL Constraints**

SQL constraints are the rules which are used to limit the type of data that is entered in a table. They are used to maintain the integrity and accuracy of the data present inside the table.

7 types of constraints – UNIQUE, NOT NULL, CHECK, PRIMARY KEY, FOREIGN KEY, DEFAULT and INDEX.

constraints can be divided into two types – Table level and Column level. The table level constraints are used to limit whole table data and the column level constraints are used to limit the column data.

**1. Table-Level Constraints**

* **Scope**: These constraints are applied to the entire table and can refer to one or more columns.
* **Purpose**: They are used when the rule involves multiple columns or when it is easier to define the constraint outside the column definitions.

**Examples:**

* **Primary Key**: Ensures that a combination of columns uniquely identifies a row in the table.

CREATE TABLE Orders (

OrderID INT,

CustomerID INT,

OrderDate DATE,

PRIMARY KEY (OrderID, CustomerID) -- **Table-level constraint**

);

* **Foreign Key**: Ensures referential integrity between two tables.

CREATE TABLE Orders (

OrderID INT,

CustomerID INT,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID) -- **Table-level constraint**

);

**2. Column-Level Constraints**

* **Scope**: These constraints are applied to a specific column.
* **Purpose**: They enforce rules on individual columns and are defined alongside the column in the table definition.

**Examples:**

* **NOT NULL**: Ensures that a column cannot have NULL values.

CREATE TABLE Customers (

CustomerID INT NOT NULL, -- **Column-level constraint**

Name VARCHAR(100)

);

**UNIQUE**: Ensures that all values in a column are distinct.

CREATE TABLE Products (

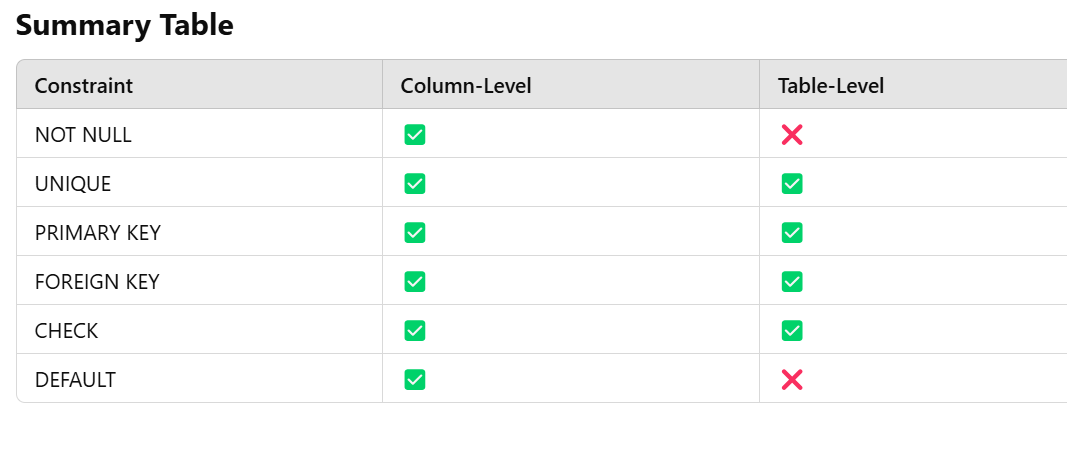
ProductID INT UNIQUE, -- **Column-level constraint**

ProductName VARCHAR(100)

);

Foreign key is a constraint used to join two different tables.

**NOT NULL** is a **column-level constraint** and **cannot** be applied at the table level.



CREATE TABLE Orders (

OrderID INT,

CustomerID INT,

OrderDate DATE,

**PRIMARY KEY** (OrderID, CustomerID), -- **Composite primary key**

**FOREIGN KEY** (CustomerID) REFERENCES Customers(CustomerID), -- **Table-level foreign key**

**CHECK** (OrderDate >= '2024-01-01') -- **Table-level check constraint**

);

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

Name VARCHAR(100) **NOT NULL,**

Age INT CHECK (Age >= 18),

Salary DECIMAL(10, 2) **DEFAULT** 50000

);

The combination of statements that can be used to add the NOT NULL constraint to an existing column is –

**UPDATE** **TABLE\_NAME**

**SET** column\_name = <value>

**WHERE** column\_name **IS** **NULL**;

&

**ALTER** **TABLE** **TABLE\_NAME**

**ALTER** **COLUMN** column\_name data\_type **NOT** **NULL**;

First, we update the table so there is no NULL in any column. Then we alter the table to change the property of the column to add the NOT NULL constraint.

**ALTER** **TABLE** **TABLE\_NAME**

**ALTER** **COLUMN** column\_name data\_type **NULL**;

To remove the NOT NULL constraint we use the ALTER command the modify the constraint of the required column and change it to NULL, so that it can accept null values.

 The statement that can be used to apply the DEFAULT constraint on a column in an existing table –

**ALTER** **TABLE** **TABLE\_NAME**

**ALTER** column\_name **SET** **DEFAULT** **VALUE**;

**ALTER** **TABLE** **TABLE\_NAME**

**ALTER** City **DROP** **DEFAULT**;

The ALTER command is used with the ALTER and the DROP keyword to delete the constraint.

**ALTER** **TABLE** **TABLE\_NAME**

**ADD** **UNIQUE** (column\_name);

To add the UNIQUE constraint to a column of an existing column, it must be used with the ALTER command.

**ALTER** **TABLE** **TABLE\_NAME**

**ADD** **CONSTRAINT** constraint\_name **UNIQUE** (column1,column2,…,columnn);

The syntax is almost same as for adding the constraint to a single column

**ALTER** **TABLE** **TABLE\_NAME**

**DROP** **INDEX** constraint\_name;

In SQL, views cannot have a check constraint

The statement that can be used to apply the table level check constraint in SQL is –

**CREATE** **TABLE** **TABLE\_NAME** (

column1 datatype **CONSTRAINT**,

column2 datatype **CONSTRAINT**,

...

columnn datatype **CONSTRAINT**,

**CONSTRAINT** constraint\_name **CHECK** (condition1 **AND** condition2));

**ALTER** **TABLE** **TABLE\_NAME**

**ADD** **CONSTRAINT** constraint\_name **CHECK** (condition);

The statement that can be used to delete the CHECK constraint from an existing table is

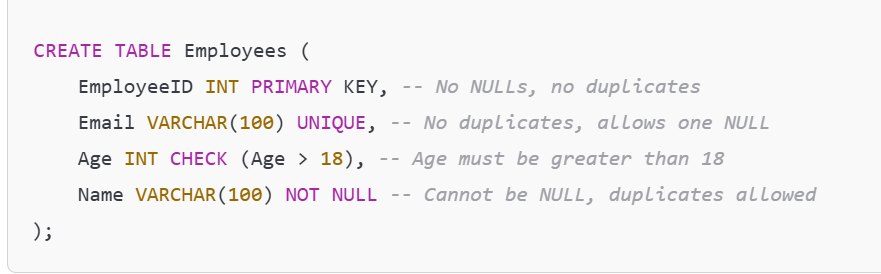
**ALTER** **TABLE** **TABLE\_NAME**

**DROP** **CHECK** constraint\_name;

**CREATE** **TABLE** employee (

Eid **VARCHAR**(10) **PRIMARY** **KEY**,

Age **INT** **CHECK** (Age **BETWEEN** 21 **AND** 40));



**CREATE** **TABLE** **TABLE\_NAME** (

column1 datatype constraint\_type,

column2 datatype constraint\_type,

...

columnn datatype constraint\_type,

**CONSTRAINT** constraint\_name **PRIMARY** **KEY** (column1, column2));

The columns that will be used to make the primary key are written within the PRIMARY KEY constraint. Constraint name is a name given to the primary key by the user.

**ALTER** **TABLE** **TABLE\_NAME**

**ADD** **PRIMARY** **KEY** (column\_name);

We use the ALTER command to add the primary key constraint to an existing table.

**ALTER** **TABLE** Persons

**DROP** **PRIMARY** **KEY**;

the FOREIGN KEY of one table is the PRIMARY KEY of another table as it is used to create a link between two tables

The table containing the foreign key is called the child table and the table containing the primary key is called the referenced or parent table.



**Parent Table**: Contains the **primary key** (e.g., Departments).

**Child Table**: Contains the **foreign key** that references the parent table (e.g., Employees).

The statement that is used to create a FOREIGN KEY on an already existing table is –

**ALTER** **TABLE** **TABLE\_NAME**

**ADD** **CONSTRAINT** constraint\_name **FOREIGN** **KEY** (column\_name)

**REFERENCES** parent\_table (column\_name);

The statement that can be used to delete the FOREIGN KEY constraint from a table is –

**ALTER** **TABLE** **TABLE\_NAME**

**DROP** **FOREIGN** **KEY** constraint\_name;

In SQL, unlike the PRIMARY KEY constraint, a table can have more than one column with the FOREIGN KEY constraint. That means that a table can be connected to more than one table. Each table’s primary key will act as a foreign key for the child table.

**The statement that can be used to apply the FOREIGN KEY constraint to more than column in a table is –**

**CREATE TABLE child\_table (**

**column1 datatype CONSTRAINT,**

**...**

**columnn datatype CONSTRAINT,**

**FOREIGN KEY (col1, col2,...) REFERENCES parent\_table (col1, col2));**

**Example with Composite Foreign Key:**

Let's create a **parent table** and a **child table** with a composite foreign key.

**Parent Table:**

CREATE TABLE parent\_table (

col1 INT NOT NULL,

col2 INT NOT NULL,

other\_column VARCHAR(100),

PRIMARY KEY (col1, col2)

);

**Child Table:**

CREATE TABLE child\_table (

child\_col1 INT NOT NULL,

child\_col2 INT NOT NULL,

other\_column VARCHAR(100),

CONSTRAINT fk\_parent\_child FOREIGN KEY (child\_col1, child\_col2) REFERENCES parent\_table (col1, col2)

);

A stored procedure is a prepared SQL code that can be saved, so that the code can be reused over and over again.

**CREATE** **PROCEDURE** procedure\_name

**AS**

Sql\_statements

**GO**;

**EXEC** procedure\_name;

A stored procedure can be called from any other stored procedure, just as we execute a stored procedure in a separate SQL query.

Stored Procedure can be available to applications accessing a relational database system. They are actually stored in the database data dictionary.

Any parameter added to a stored procedure must be preceded with @ symbol and must end with the data type of the parameter.

**CREATE** **PROCEDURE** money @note varchar2(10)

**AS**

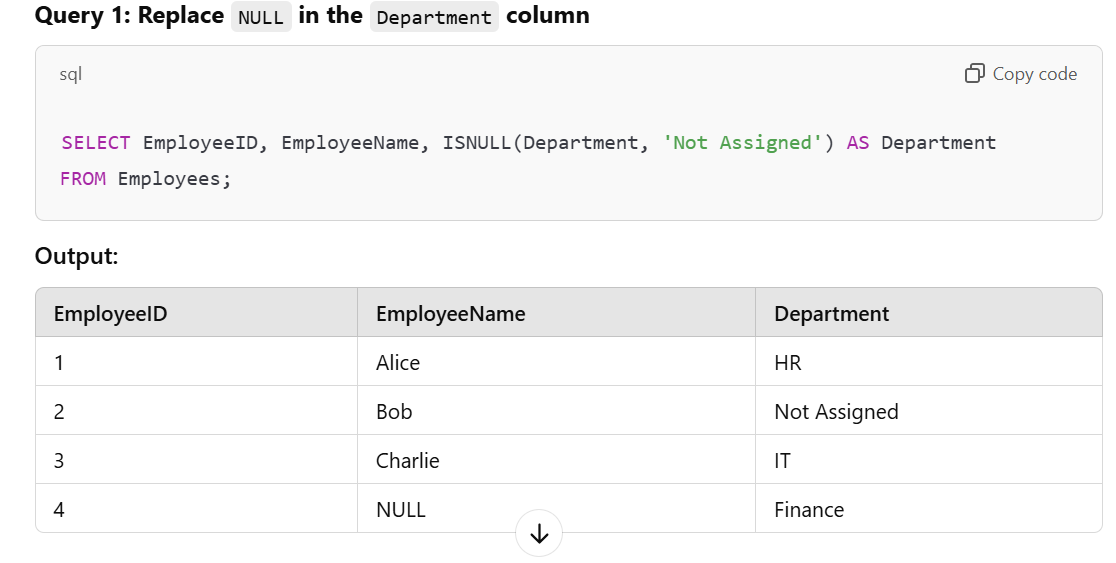
Sql\_statements

**GO**;

In SQL server ISNULL ( ) function is used to replace the NULL values with some other value.

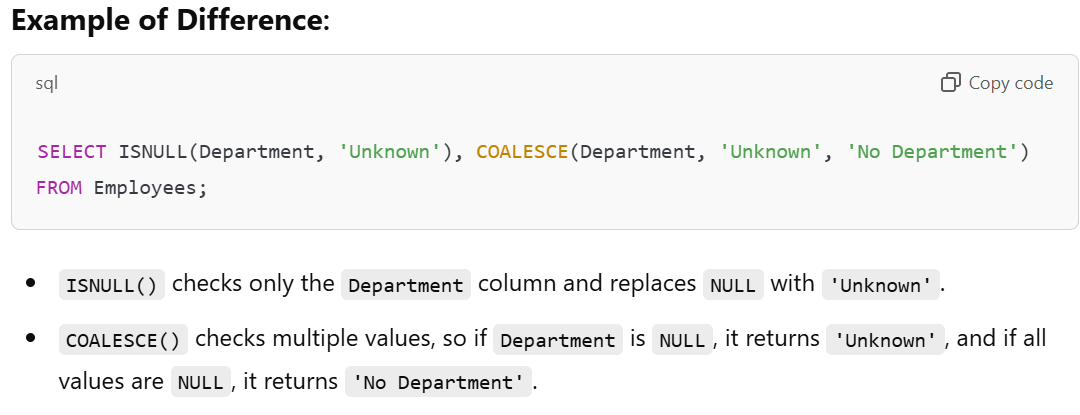
SELECT column\_list,

ISNULL(column\_name, replace\_value) FROM table\_name;



ISNULL ( ) function is used to test whether an expression is NULL or not. If the expression is NULL it returns TRUE, else FALSE.

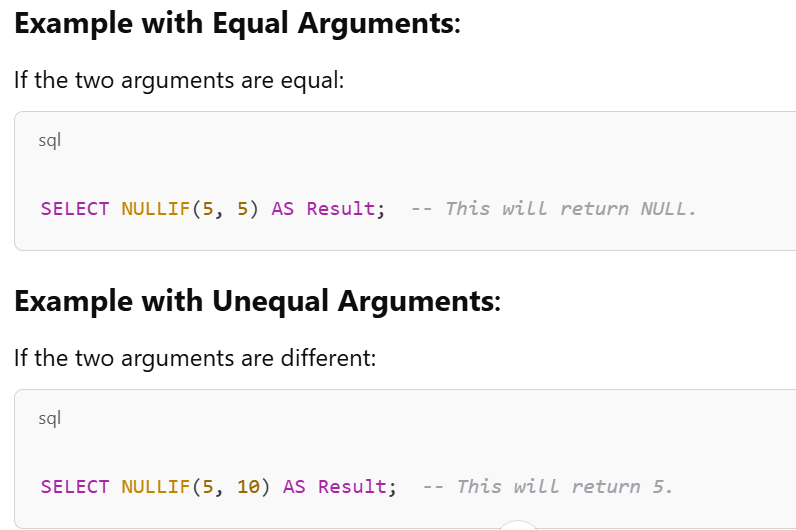
COALESCE ( ) function in SQL returns the first non-NULL expression or value among its arguments.



The COALESCE ( ) function returns the first expression with non-null value from a list. If all the expressions are evaluated to be null, then the COALESCE ( ) function will return null.

**SELECT** column\_list, **COALESCE**(expressions\_list)

**FROM** **TABLE\_NAME**;



**SELECT** column\_list, **NULLIF** (exp1, exp2)

**FROM** **TABLE\_NAME**;

NVL ( ) function is used to replace NULL value with another value. It is similar to the IFNULL function in MySQL and the ISNULL function in SQL Server.

Data of a table can be copied and inserted into another table in SQL by making use of INSERT INTO SELECT statement.

INSERT INTO employees\_backup (id, name, position, salary)

SELECT id, name, position, salary

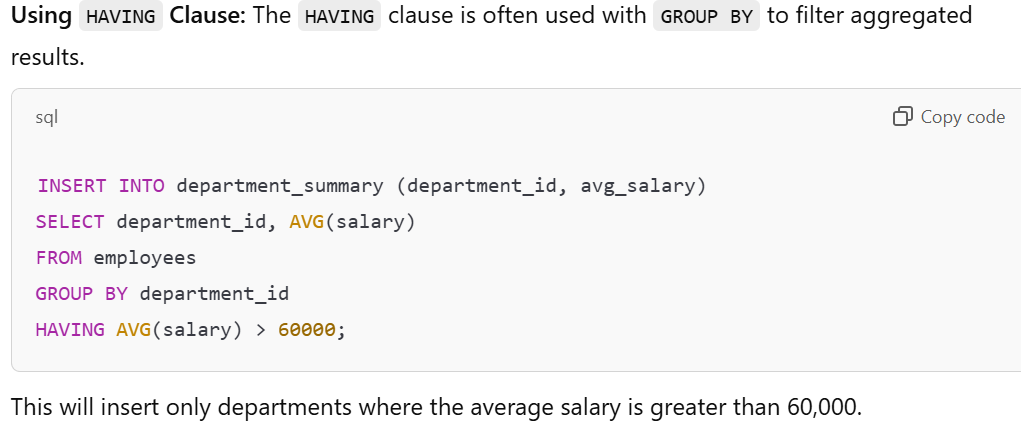
FROM employees;

The existing records of the target table of INSERT INTO SELECT statement remain unaffected. The new records from source table will be inserted into the target table without affecting the previous records.

**INSERT** **INTO** target\_table

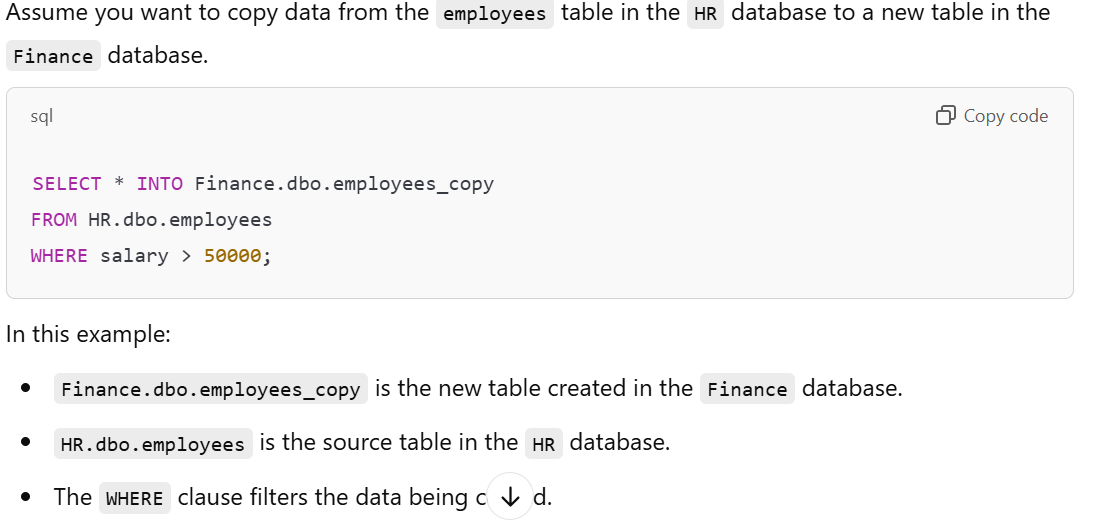
**SELECT** \* **FROM** source\_table

**WHERE** condition;

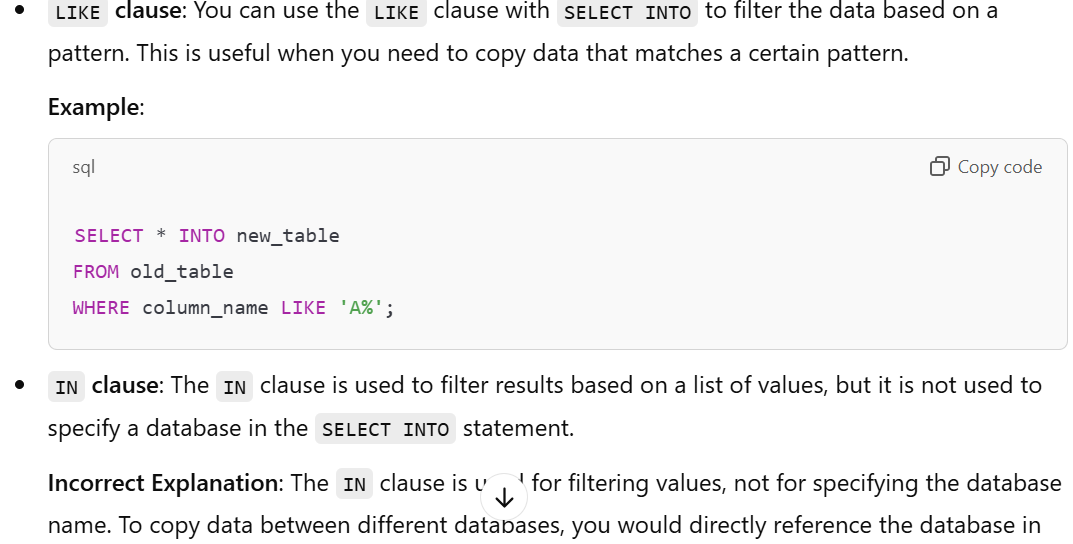


The SELECT statement of INSERT INTO SELECT can easily contain WHERE, GROUP BY, and HAVING clauses, as well as table joins and aliases.

WHERE condition is optional to use along with SELECT INTO Clause. Using WHERE clause limits the number of records to copy from a table based on the given conditions.



SELECT INTO will create a new table but **does not copy constraints, indexes, or triggers** from the source table. If you need to include them, you would need to create them manually after copying the data.



 If the condition in WHERE clause of a SELECT INTO statement returns no data, then \_\_\_\_\_\_  
c) An empty new table will be created

SELECT INTO statement can be used to create a new, empty table using the schema of another. We should use the WHERE clause which doesn’t return any data to create a new table.

SELECT INTO statement can be used to create a new empty table. We need to add a WHERE clause that causes the query to return no data.

**SELECT** \* **INTO** newtable

**FROM** oldtable

**WHERE** 3=4;

New table used in SELECT INTO statement should be the one which doesn’t exist before. If it does already exist, the SELECT INTO statement will raise an error.

