Introduction to SQL:

SQL is a standard language for accessing and manipulation of database.

★ What is SQL ?

- it stands for structured Query Language
- it lets us access and manipulate databases.
- it became a standard of American National standard Institute (ANSI) in 1986, and international Organization for standardization (ISO) in 1987.

★ What can SQL DO?

- SQL can execute queries against a database.
- " " retreive data from a database.
- " " insert records in " " .
- " " update " " " " .
- " " Delete " " " " .
- SQL can create new databases.
- SQL can create new Tables in a database.
- SQL can create stored Procedures in a database.
- SQL can create views in database.
- SQL can set Permission on Tables, Procedures and views.

Tables:

A Table is a collection of related data entries and it consists of columns and rows.

① Fields: it refers to column in a Table, & it is designed To maintain specific information about every record in The Table. it contains all information associated with a specific field in a Table.

② Record: it refers To row in a Table, and it holds data about each individual data.

★ Semicolom after SQL statements?

Some database systems require a semicolom at the end of each SQL statement.

Semi colom is the standard way to separate each SQL statement in database systems that allow more than one SQL statements to be executed in the same call to the server.

NOTE: SQL Keywords are NOT case sensitive: select is the same as SELECT

The Most Important SQL Commands:

- SELECT - extracts data from a database.
- UPDATE - updates data in a database
- DELETE - deletes data from a database
- INSERT INTO - inserts new data into a database
- CREATE DATABASE - creates a new database
- ALTER DATABASE - modifies a database
- CREATE TABLE - creates a new Table
- ALTER TABLE - modifies a Table
- DROP TABLE - deletes a Table
- CREATE INDEX - creates an index (search Key)
- DROP INDEX - deletes an index

P.T.O...

SQL Select Statement:

The "SELECT" statement is used to select data from a database. The data returned is stored in a result Table, called The result-set.

SYNTAX:-

```
SELECT COLUMN_NAME_1, COLUMN_NAME_2, COLUMN_NAME_3 FROM Table_Name;
```

(here column_Name_1, 2, 3 are the field names of the Table we want to select data from)

* IF we want to SELECT All the fields Available in Table:

```
SELECT * FROM Table_name;
```

SQL SELECT DISTINCT Statement:

The "SELECT DISTINCT" statement is used to return only distinct (different values). Inside a table, a column often contains many duplicate values; and sometimes we need only the list of distinct (or different) values.

SYNTAX:-

```
SELECT DISTINCT COLUMN1, COLUMN2, ... FROM Table_name;
```

* IF we want to count the Distinct entries for a Column:

```
SELECT COUNT (DISTINCT COUNTRY) FROM Tourist_Table_Name;
```

OR we can also write it as... (or alias name)

```
SELECT COUNT (*) AS DISTINCT Column_name FROM (
    SELECT DISTINCT COUNTRY FROM Customers);
```

SQL WHERE CLAUSE:

The "WHERE" CLAUSE is used to filter records.

It is used to extract only those records that fulfill a specific condition.

SYNTAX:-

```
SELECT column_name FROM table_name WHERE condition;
```

NOTE: The "WHERE" clause is not only used "SELECT" statements, it is also used in "UPDATE", "DELETE", etc.!

eg: `SELECT * FROM CUSTOMERS WHERE COUNTRY = 'Mexico';`

it would return all the records (rows) which have country as mexico.

Text fields Vs Numeric fields:

SQL requires single quotes around text values (sometimes Double quotes).

However, numeric fields aren't enclosed in quotes.

OPERATORS IN The WHERE CLAUSE:

OPERATOR	DESCRIPTION	EXAMPLE
'='	Equal	• SELECT * FROM PRODUCTS WHERE PRICE = 18;
'>'	GREATER Than	• SELECT * FROM PRODUCTS WHERE Price>
'<'	LESS Than	• " " " " " Price < 30;
'>='	GREATER Than or equal	• " " " " " Price ≥ 30;
'<=' OR	LESS Than or equal	• " " " " " Price ≤ 30;
'<>' OR '!='	NOT equal	• " " " " " Price <> 30;
'BETWEEN'	BETWEEN a Certain range	• " " " " where Price BETWEEN 50 AND 60;
'LIKE'	Search for a Pattern	• SELECT * FROM CUSTOMER WHERE city LIKE 's%'; (returns all cities starting with s)
'IN'	To specify Multiple Possible Values for a column	• SELECT * FROM CUSTOMER WHERE City IN ('Paris', 'LONDON'); (returns customer who belong to city Paris, LONDON)

SQL AND, OR and NOT Operators:

The "where" clause can be combined with AND, OR, and NOT operators.

The AND and OR operators are used to filter records based on more than one condition.

* The AND operator displays a record if all the conditions separated by AND are TRUE.
* The OR operator displays a record if any of the conditions separated by OR is TRUE.
* The NOT operator displays a record if the condition is not TRUE.

AND Syntax:

```
SELECT column1, column2... FROM Table_name WHERE CONDITION1
AND CONDITION2 AND CONDITION3... ;
```

OR SYNTAX:

```
SELECT column1 FROM Table_name WHERE CONDITION1 OR CONDITION2;
```

NOT SYNTAX:

```
SELECT column1, column2... FROM table_name WHERE NOT CONDITION;
```

Examples:

* `SELECT * FROM CUSTOMER WHERE COUNTRY = 'GERMANY' AND CITY = 'MEXICO';`
* `SELECT * FROM CUSTOMER WHERE COUNTRY = 'Germany' OR COUNTRY = 'BERLIN';`
* `SELECT * FROM CUSTOMER WHERE NOT COUNTRY = 'GERMANY';`

COMBINING AND, OR and NOT:

eg: SELECT * FROM CUSTOMER WHERE COUNTRY = 'GERMANY' AND (CITY = 'BERLIN' OR CITY = 'MUNCHEN');

eg: SELECT * FROM CUSTOMER WHERE NOT COUNTRY = 'GERMANY' AND NOT COUNTRY = 'USA';

SQL ORDER BY KEYWORD:

The 'ORDER By' Keyword is used To sort The result set in ascending or descending order.

The 'ORDER By' Keyword sorts The records in ascending order by default. To sort The records in Descending order, we use The DESC Keyword.

★ SYNTAX:

SELECT column1, column2,... FROM table_name ORDER By column1, column2,... ASC/DESC;

① eg: To Sort in Ascending Order:

SELECT * FROM CUSTOMER ORDER By COUNTRY;

② eg: To Sort in Descending Order:

SELECT * FROM CUSTOMER ORDER By COUNTRY DESC;

③ eg: To use ORDER By on SEVERAL columns:

SELECT * FROM CUSTOMER ORDER By COUNTRY, CUSTOMER NAME;

(Explanation: This means it orders By COUNTRY, But if some rows have same COUNTRY, it Then orders Them by Customer names.)

④ eg: To USE ORDER By on SEVERAL Columns But in Different ORDERS:

SELECT * FROM CUSTOMER ORDER By COUNTRY ASC, CUSTOMER NAME DESC;

(Explanation: The logic remains same as Previous query just order of sorting changes.)

SQL INSERT INTO STATEMENT:

The "INSERT INTO" statement is used to insert new records in the table.

SYNTAX:

① INSERT INTO table_name (column1, column2, ...) Values (value1, value2, value3 ....);

(here we specified both the column names and the values to be inserted.)

②. INSERT INTO table_name VALUES (value1, value2, value3 ...);

(if we are adding values for all the columns of table, then we do not need to specify column name in the query. However, we have to make sure the order of the values is in the same order as the columns in the table)

★ Examples:

INSERT INTO CUSTOMERS (CUSTOMER NAME, CONTACT NAME, ADDRESS, CITY, POSTAL CODE, COUNTRY) VALUES ('CARDINAL', 'TOM B. Erichsen', 'SKAGEN 21', 'STAVANGER', '4006', 'NORWAY');

★ Example: TO INSERT DATA ONLY IN SPECIFIED COLUMNS

INSERT INTO CUSTOMERS (CUSTOMER NAME, CITY, COUNTRY) Values ('Cardinal', 'Stavanger', 'Norway');

// The other columns would be filled with Null value

SQL NULL VALUES:

A field with a "NULL VALUE" is a field with NO value. if a field in a table is optional, it is possible to insert a new record or update a record without adding a value to that field. Then the field will be saved with a NULL value.

NOTE: A NULL VALUE IS DIFFERENT FROM A zero value or a field that contains spaces.

A field with NULL value is one that has been left blank during record creation.

To Test for NULL VALUES:

Here we use "IS NULL" and "IS NOT NULL" OPERATORS.

★ IS NULL SYNTAX:

```
SELECT column_names FROM table_name WHERE column_name
IS NULL;
```

★ IS NOT NULL SYNTAX:

```
SELECT column_names FROM table_name WHERE column_name
IS NOT NULL;
```

Examples:

① SELECT CUSTOMERNAME, ContactName, Address FROM CUSTOMERS WHERE Address IS NULL;

② SELECT CUSTOMERNAME, CONTACTNAME, ADDRESS FROM CUSTOMER WHERE Address IS NOT NULL;

SQL UPDATE STATEMENT:

The "UPDATE STATEMENT" is used to modify existing records in a table.

SYNTAX:

```
UPDATE table_name SET column1 = value1, column2 = value2, ...
where condition;
```

[NOTE: Here the Where clause specifies which records should be updated.

if we omit the where clause, all records in the table will be updated.]

changes

* Example:

```
UPDATE CUSTOMERS SET CONTACT NAME = 'Alfred Schmidt', CITY = 'FRANKFURT
Where CUSTOMER ID = 1;
```

UPDATE MULTIPLE RECORDS:

It is "where" clause that determines how many records will be updated.

* eg: UPDATE CUSTOMER SET CONTACT NAME = 'JUAN' WHERE COUNTRY = 'Mexico';

[NOTE: Here if we Omit the Where clause, All the records of ATTRIBUTE (CONTACTNAME) would Be updated To "JUAN"]

SQL DELETE STATEMENT:

The "DELETE" STATEMENT is used TO DELETE EXISTING RECORDS IN A TABLE.

* SYNTAX:

```
DELETE FROM table-Name Where condition;
```

[NOTE: Here the where clause specifies which records should be deleted. if we "Omit the Where clause", all records in the Table will be DELETED.]

* Example:

```
DELETE FROM CUSTOMERS WHERE CUSTOMERNAME = 'Alfred';
```

DELETING ALL RECORDS:

it is POSSIBLE TO delete all rows in a Table without deleting the Table.

This means that the Table structure, attributes, and indexes will be intact.

* SYNTAX:

```
DELETE FROM table_name;
```

SQL TOP, LIMIT, FETCH FIRST OR ROWNUM CLAUSE:

The "SELECT TOP" clause is used to specify the number of records to return.

The SELECT TOP clause is USEFUL ON LARGE TABLES with thousands of records.

[NOTE: Not all database systems support the SELECT TOP clause. My SQL SUPPORTS the LIMIT CLAUSE to select a limited number of records. while ORACLE USES FETCH FIRST n ROWS ONLY and ROWNUM.]

★ SYNTAX FOR SQL SERVER/ MS ACCESS Syntax:

```
SELECT TOP NUMBER\PERCENT COLUMN_Name FROM Table_name
WHERE CONDITION;
```

eg: SELECT TOP 3 * FROM CUSTOMERS;

★ SYNTAX FOR My SQL:

```
SELECT Columm_name FROM table_name WHERE condition LIMIT
number;
```

eg: SELECT * FROM CUSTOMERS LIMIT 3;

★ ~~ORACLE 12 SYNTAX:~~

~~SELECT column_name FROM table_name WHERE ROWNUM <= number;~~

~~eg: SELECT * FROM CUSTOMER WHERE RowNum <= 3;~~

★ OLDER ORACLE SYNTAX:

```
SELECT Columm_name FROM table_name WHERE ROWNUM ≤ number;
```

★ ORACLE 12 SYNTAX:

```
SELECT Columm_name FROM table_name ORDER BY Columm_name
FETCH FIRST NUMBER ROWS Only;
```

eg: SELECT * FROM CUSTOMER FETCH FIRST 3 ROWS ONLY;

★ OLDER ORACLE SYNTAX (with ORDER BY) SYNTAX:

SELECT * FROM (SELECT column_name(s) FROM table_name ORDER BY column_name(s)) WHERE ROWNUM <= number;

SQL TOP PERCENT EXAMPLES:

eg: SELECT TOP 50 PERCENT * FROM CUSTOMERS;

eg: SELECT * FROM CUSTOMERS FETCH FIRST 50 PERCENT ROWS ONLY;

ADDING A WHERE CLAUSE:

eg: SELECT TOP 3 * FROM CUSTOMERS WHERE COUNTRY = 'GERMANY';

eg: SELECT * FROM CUSTOMERS WHERE COUNTRY = 'GERMANY' LIMIT 3;

eg: SELECT * FROM CUSTOMERS WHERE COUNTRY = 'GERMANY' FETCH FIRST 3 ROWS ONLY;

SQL MIN() and MAX() FUNCTIONS:

The "Min()" function returns the smallest value of selected column.

The "Max()" function returns the largest value of the selected column.

★ MIN() SYNTAX:

SELECT MIN(column_name) FROM table_name where condition;

eg: SELECT MIN(PRICE) AS smallest price FROM products;
(Alias)

★ MAX() SYNTAX:

SELECT MAX(column_name) FROM table_name where condition;

eg: SELECT MAX(PRICE) AS LARGEST PRICE FROM PRODUCTS;

SQL COUNT(), AVG() and SUM() FUNCTIONS:

★ The "COUNT()" function returns the number of rows that matches a specified criterion.

SYNTAX:

SELECT COUNT (column_name) FROM table_name WHERE condition;

examples:

SELECT COUNT (PRODUCT ID) FROM PRODUCTS;

[NOTE: NULL values are not counted.]

★ The "AVG()" function returns the average value of a numeric column.

Syntax:

SELECT AVG (column_name) FROM table_name Where condition;

eg:

SELECT AVG (PRICE) FROM PRODUCTS;

[NOTE: NULL values are ignored.]

★ The "SUM()" function returns the total sum of a numeric column.

SYNTAX:

SELECT SUM (column_name) FROM table_name WHERE condition;

eg:

SELECT SUM (QUANTITY) FROM OrderDetails;

[NOTE: NULL Values are ignored.]

T.T.D...

SQL LIKE OPERATOR:

The "LIKE" OPerator is used in where clause to search for specified pattern in a column. There are Two wild cards often used in conjuction with the LIKE OPERATOR.

- The Percent sign (%) represents Zero, one, or MULTIPLE Characters.
- The underscore sign (_) represents one, single character.

[NOTE: MS Access uses an asterik (*) instead of the Percent sign (%) and a quetion mark (?) instead of the underscore (_).]

SYNTAX:

```
SELECT column1, column2, .... FROM table_name WHERE Column
 LIKE PATTERN;
```

LIKE OPERATOR WITH '%' and '_' wild cards:

LIKE OPERATOR	DESCRIPTION
• WHERE CUSTOMERNAME LIKE 'a%'	find any values that start with 'a'
• WHERE CUSTOMER NAME LIKE '%a'	find any values that end with 'a'.
• WHERE CUSTOMERNAME LIKE '%or%'	find any values that have "or" in any Position.
• WHERE CUSTOMERNAME LIKE '_r%'	find any values that have "r" in second Position.
• WHERE CUSTOMER NAME LIKE 'a_%'	finds any values that starts with "a" and are atleast 2 characters in length.
• WHERE CUSTOMERNAME LIKE 'a__%'	finds any values that starts with "a" amd are atleast 3 characters in length.
• WHERE CONTACT NAME LIKE 'a%o'	finds any values that starts with "a" and ends with "o".

★ Examples:

- SELECT * FROM CUSTOMERS WHERE CUSTOMERNAME LIKE 'a%';

(The following SQL statement selects all customers with CustomerName starting with "a").

- SELECT * FROM CUSTOMERS WHERE CUSTOMERNAME LIKE '%a';

(The following SQL statement selects all customers with CustomerName ending with "a").

- SELECT * FROM CUSTOMERS WHERE CUSTOMERNAME LIKE '%or%';

(The following SQL statement selects all customers with a CustomerName that have "or" in any Position).

- SELECT * FROM CUSTOMERS WHERE CUSTOMERNAME NOT LIKE 'a%';

(The following SQL statement selects all customers with a CustomerName that doesnot start with "a").

SQL WILDCARDS Charcters:

A "WILD CARD" Character is used to substitute one or more characters in a string.

Wildcard characters are used with the "LIKE operator". The LIKE OPERATOR is used in a "WHERE" CLAUSE to search for a specified Pattern in a column.

★ WILD CARD CHARACTERS IN SQL SERVER:

SYMBOL	DESCRYPTION	Example
%	Represents zero or more characters.	bl% would find black, blue and blob.
_	Represents a single character	h_t would find hot, hat, hit.
[]	Represents any single character within brackets	h[oa]t would find hot, hat but not hit as i not present in bracket.
^	Represents any character not in brackets.	h[^oa]t would find hit (& not hot or hat)
–	Represents any single character within specified range	c[a-b]t finds cat

★ Examples: ★ USING The "%" Wildcard:

- The following SQL statement selects all Customers with a city starting with "ber":

```
SELECT * FROM CUSTOMERS WHERE CITY LIKE 'ber%';
```

- The following SQL statement selects all Customers with a city containing the Pattern "es":

```
SELECT * FROM CUSTOMERS WHERE CITY LIKE '%es%';
```

★ USING The "_" WILDCARD:

```
SELECT * FROM CUSTOMERS WHERE CITY LIKE '_ondon';
```

(The above statement selects all customers with a city starting with any character, followed by "ondon".)

- The following SQL statement selects all customers with a city starting with "L", following by any character, followed by "m", followed by any character, followed by "on":

```
SELECT * FROM CUSTOMERS WHERE City LIKE "L-m-on";
```

★ USING The [charlist] WILDCARD:

- The following SQL statement selects all customers with a City starting with "b", "s" or "p":

```
SELECT * FROM CUSTOMERS WHERE CITY LIKE '[bsp]%';
```

- The following SQL statement selects all customers with a city starting with "a", "b" or "c"

```
SELECT * FROM CUSTOMERS WHERE CITY LIKE '[a-c]%';
```

★ USING The [!charlist] Wildcard:

- The Two following SQL statements select all customers with a city NOT starting with "b", "s" or "p"

```
SELECT * FROM CUSTOMERS WHERE CITY LIKE '[!bsp]%';
```

Or

```
SELECT * FROM CUSTOMER WHERE CITY NOT LIKE '[bsp]%';
```

SQL IN OPERATOR:

The "IN" operator allows you to specify multiple values in "where" clause.

The "IN" OPERATOR is a shorthand for multiple 'OR' conditions

★ SYNTAX:

- SELECT Column_name(s) FROM table_name WHERE Column_name IN (value 1, value 2, ...);

OR

- SELECT column_name(s) FROM table_name WHERE Column_name IN (SELECT statement);

★ EXAMPLE:

- The following SQL statement selects all customers that are located in "GERMANY", "FRANCE" OR "UK".

```
SELECT * FROM CUSTOMERS WHERE COUNTRY IN ('GERMANY',
'France', 'UK');
```

- The following SQL statement selects all customers that are NOT LOCATED in "GERMANY", "FRANCE" OR "UK".

```
SELECT * FROM CUSTOMERS WHERE COUNTRY NOT IN (
'GERMANY', 'France', 'UK');
```

- The following SQL statement selects all customers that are from same country as the suppliers:

```
SELECT * FROM CUSTOMERS WHERE COUNTRY IN (SELECT
COUNTRY FROM SUPPLIERS);
```

P.T.O...

SQL BETWEEN OPERATOR:

The "BETWEEN" OPERATOR SELECTS VALUES within a given range. The values can be number, text or dates.

The Between operator is inclusive : ie begin & end values are included.

★ SYNTAX:

```
SELECT column_name(s) FROM table_name WHERE
column_name BETWEEN Value 1 AND value 2;
```

★ EXAMPLE:

```
SELECT * FROM PRODUCTS WHERE Price Between 10 AND 20;
```

★ NOT BETWEEN EXAMPLE:

```
SELECT * FROM PRODUCTS WHERE price NOT BETWEEN 10 AND 20;
```

★ BETWEEN with IN EXAMPLE:

```
SELECT * FROM PRODUCTS WHERE Price BETWEEN 10 AND 20
AND CATEGORY_ID NOT IN (1, 2, 3);
```

★ BETWEEN TEXT VALUES:

```
SELECT * FROM Products WHERE ProductName BETWEEN 'BEARDO'
AND 'MAMA EARTH' ORDER By Product Name;
```

★ BETWEEN DATES EXAMPLE:

The following SQL statement selects all orders with an OrderDate between '01-July-1996' and '31-July-1996'.

Example:

```
SELECT * FROM Orders WHERE OrderDate BETWEEN #07/01/1996#
AND #07/31/1996#;
```

OR

```
SELECT * FROM Orders WHERE OrderDate BETWEEN '1996-07-01'
AND '1996-07-31';
```

SQL ALIASES:

SQL aliases are used to give a table, or a column in a table a temporary name.

Aliases are often used to make column name more readable. An Alias only exists for the duration of that query.

An Alias is created with the "AS" Keyword.

★ Alias Column Syntax:

```
SELECT column_name AS alias_name FROM table_name;
```

eg: The following SQL statement creates two aliases, one for the CustomerID column and one for the CustomerName column.

```
SELECT CUSTOMERID AS ID, CUSTOMERNAME AS CUSTOMER FROM
CUSTOMERS;
```

★ Alias Table Syntax:

```
SELECT column_name(s) FROM table_name AS alias_name;
```

Some Examples:

★ The following SQL statement creates two aliases, one for the CustomerName column and one for the ContactName column.

```
SELECT CUSTOMERNAME AS CUSTOMER, ContactName AS [CONTACT PERSON]
FROM customers;
```

[NOTE: it requires double quotation marks or square brackets if the alias name contains spaces.]

★ The following SQL statement creates an alias named "ADDRESS" that combine four columns (Address, PostalCode, City and Country)

```
SELECT CUSTOMERNAME, ADDRESS + ',' + PostalCode + ' ' + City + ',' +
COUNTRY AS ADDRESS FROM CUSTOMERS;
```

OR

```
SELECT CUSTOMERNAME, CONCAT(Address, ',', PostalCode, ',', City, ',',
COUNTRY) AS ADDRESS FROM CUSTOMERS;
```

OR

```
SELECT CUSTOMERNAME, (ADDRESS || ',' || PostalCode || ',' || City || ',' || COUNTRY) AS
ADDRESS FROM CUSTOMERS;
```

* Alias for Table Syntax:

SELECT o.orderID, o.OrderDate, c.customerName FROM Customers AS c, Orders AS o WHERE c.customerName = 'Around The horn' AND c.customerID = o.customerID;

(In the above query, it selects all the orders from the Customer with CustomerID = 4 (i.e Around The horn). We use the "Customers" and "Orders" Tables, and give them the Table aliases of "c" and "o" respectively (Here we use aliases to make SQL shorter)

if we write same query But without alias then.

SELECT Orders.OrderID, Orders.OrderDate, CUSTOMERS.CUSTOMERName FROM Customers, Orders WHERE CUSTOMERS.CUSTOMERNAME = 'Around The horn' AND Customers.CustomerID = Orders.CustomerID;

Aliases can be useful when:

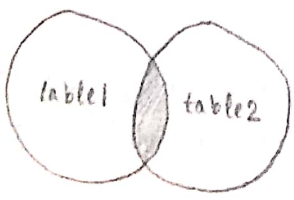
- There are more than one table involved in a query.
- FUNCTIONS are used in the query.
- Column names are big or not very readable
- Two or more columns are combined together

SQL JOIN:

A "JOIN" clause is used to combine rows from two or more tables, based on a related column between them.

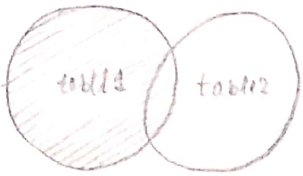
1) INNER JOIN: That Selects records that have matching values in both Tables.

eg: SELECT Orders.OrderID, CUSTOMERS.customerName, Orders.OrderDate FROM ORDERS INNER JOIN CUSTOMERS ON Orders.CustomerID = Customer.customerID;



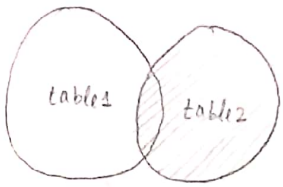
INNER JOIN

2) LEFT (OUTER) JOIN: Returns all records from left table, and the matched records from the right table.



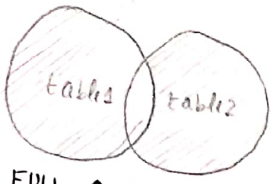
LEFT OUTER JOIN

3) Right (OUTER) JOIN: Returns all records from right table, and matched records from left table.



RIGHT JOIN

4) FULL (OUTER) JOIN: Returns all records when their is match in either left or right Table



FULL OUTER JOIN

SQL INNER JOIN:

The "Inner Join" Keyword selects records that have matching values in both Table.

★ SYNTAX:

```
SELECT column_table(s) FROM table1 INNER JOIN table2 ON
table1.column_name = table2.column_name;
```

eg: SELECT Orders.OrderID, Customers.CustomerName FROM Orders INNER JOIN CUSTOMERS ON Order.customerID = customer.customerID;

NOTE: The Inner Join Keyword selects all rows from both Tables as long as there is a match between columns.

if There are records in The "orders" table That do not have matches in "Customers", These orders will not be shown.

SQL LEFT JOIN:

The "Left JOIN" Keyword returns all records from the left Table (i.e table1) and the matching records from right Table (i.e table2).

The result is 0 records from right side, if there is no match.

★ Syntax:

```
SELECT Column_name FROM table1 LEFT JOIN table2 ON table1.column_name
  = table2.column_name ;
```

★ NOTE: The Left JOIN Keyword returns all records from left Table, even if there are no matches in the right Table (orders)

SQL Right JOIN:

The "Right JOIN" Keyword returns all records from the right Table (table2), and matching records from the left table (table1). The result is 0 records from left side, if there is no match.

★ Syntax:

```
SELECT column_name(s) FROM table1 Right JOIN table2 ON
  table1.column_name = table2.column_name ;
```

★ NOTE: The right JOIN (even) keyword returns all records from right table, if there are no matches in left table.

SQL FULL OUTER JOIN:

The "FULL OUTER JOIN" Keyword returns all records when Their is a match in a left (table1) or right (table2) table records.

★ Syntax:

```
SELECT column_name(s) FROM table1 FULL OUTER JOIN table2 ON
  table1.column_name = table2.column_name where condition;
```

★ NOTE:
it would select all the records from both Table That would satisfy where condition. i.e it returns all matching records from both tables when other Tables matches or not. so, if There are rows in "customers" That do not matches in "Orders" or vice versa. Then all Those rows will be listed as well.

SQL self JOINS

A "SELF JOIN" is a regular JOIN, but the table is JOINED with ITSELF.

★ SYNTAX:

SELECT column_name(s) FROM table1 T1, table1 T2 WHERE condition;

SQL UNION Operator:

The "UNION" operator is used to combine the result set of two or more "SELECT" statements.

- EVERY SELECT statement within UNION must have the same number of columns.
- The columns must also have similar data types
- The columns in every SELECT statement must also be in same order.

★ UNION SYNTAX:

SELECT column_name(s) FROM table1 UNION SELECT column_name(s) FROM table2;

[NOTE: The union operator selects only Distinct values by default.
To allow duplicate values, we use UNION ALL.]

★ UNION ALL SYNTAX:

SELECT column_name(s) FROM table1 UNION ALL SELECT column_name(s) FROM table2;

SQL GROUP BY Statement:

The "GROUP BY" statement groups rows that have the same values into summary rows, (like "find the no. of customers in each country").

The "GROUP BY" statement is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns.

★ GROUP BY SYNTAX:

SELECT column_name(s) FROM table_name WHERE condition
GROUP BY column_name(s) ORDER BY column_name(s);

★ examples:

SELECT COUNT (CUSTOMER ID), COUNTRY FROM CUSTOMERS GROUP BY COUNTRY;

The above SQL statement lists the number of customers in each country.

SQL HAVING CLAUSE:

The "HAVING" clause was added to SQL because "where" keyword cannot be used with aggregated functions.

★ HAVING SYNTAX:

SELECT column_name(s) FROM table_name where condition GROUP BY column_name(s) HAVING CONDITION ORDER BY column_name(s);

★ Examples:

① SELECT COUNT (CUSTOMER ID), country FROM Customers GROUP BY COUNTRY HAVING COUNT (CustomerID) > 5;

(it lists the number of customers in each country. only include countries with more than 5 customers).

② SELECT COUNT (CustomerID), country FROM Customers GROUP BY COUNTRY HAVING COUNT (CustomerID) > 5 Order By COUNT (CustomerID) DESC;

(it lists the no. of customers in each country, sorted high to low (only include countries with more than 5 customers)

A.T.

SQL EXISTS Operator:

The "Exists" operator is used to test for the existence of any record in a subquery.

The "Exists" operator returns True if the subquery returns one or more records.

★ Syntax:

```
SELECT column_name(s) FROM table_name WHERE EXISTS
(SELECT column_name FROM table_name WHERE condition);
```

★ Examples:

```
SELECT SupplierName FROM Suppliers WHERE EXISTS
(SELECT ProductName FROM Products WHERE Products.SupplierID =
  Supplier.SupplierID AND Price < 20);
```

(The following SQL statement returns TRUE and LISTS the suppliers with a product Price less than 20)

SQL ANY and ALL Operators:

The "ANY" and "ALL" operator allows us to Perform a comparison b/w a single column value and a range of other values.

SELECT INTO SYNTAX:

The "SELECT INTO" statement copies data from one table into a new table.

★ SYNTAX:

```
SELECT * INTO newtable [IN externaldb] FROM oldtable WHERE
condition;
```

(it would copy some columns into table)

eg:

```
SELECT * INTO CUSTOMER GERMANY FROM CUSTOMERS where
Country = 'Germany';
```

The SQL INSERT INTO SELECT STATEMENT:

The "INSERT INTO SELECT" statement copies data from one table and inserts it into another table.
(it requires that data types in source & target tables match.)

[NOTE: The existing records in the target table are unaffected]

★ Syntax:

[To copy all columns from one table to another table:]

```
INSERT INTO table2  SELECT * FROM table1 WHERE condition;
```

[To copy some columns from one table into another table.]

```
INSERT INTO table2 (column1, column2, ...) SELECT column1,
column2, ... FROM table1 where condition;
```

The SQL CASE Expression:

The "CASE" expression goes through conditions and returns a value ~~within~~ when the first condition is met (like an if-then-else statement).
So once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in else clause.
If there is no ELSE part & no condition are true, it returns Null

★ SYNTAX:

```
CASE
    when  condition 1   then result 1
    when  condition 2   then result 2
    when  condition 3   then result 3
    else  result
End;
```

Example:

```
SELECT OrderID, QUANTITY, CASE WHEN QUANTITY > 30 THEN 'The
quantity is greater than 30' WHEN QUANTITY = 30 THEN 'The Quantity
is 30' ELSE 'The Quantity is under 30' END AS Quantity Text
FROM Order Details;
```

SQL Create Database:

The "Create Database" statement is used to create a new SQL database.

★ Syntax:

```
CREATE DATABASE databasename;
```

★ Tip:

Make sure you have admin privilege before creating any database. Once a database is created, you can check it in the list of databases with the following SQL command:

```
SHOW DATABASES;
```

SQL DROP Database:

The "DROP Database" statement is used to drop an existing SQL Database.

★ Syntax:

```
DROP DataBase databasename;
```

SQL BACKUP Database:

The "BACKUP DATABASE" statement is used in SQL server to create a full back up of an existing SQL database.

★ Syntax:

```
BACKUP DATABASE databasename TO DISK = 'File Path';
```

Extension: SQL BACKUP with Differential statement:

A differential backup only backs up the parts of the database that have changed since last full database backup.

★ Syntax:

```
BACKUP DATABASE databasename TO DISK = 'file Path' with DIFFERENTIAL;
```

SQL CREATE Table:

The CREATE TABLE statement is used to create a new table in a database.

★ Syntax:

```
CREATE TABLE table_name ( column1, datatype, column2 datatype... );
```

★ Example:

```
CREATE TABLE Persons ( PersonID int, LastName varchar (255),
                       FirstName varchar (255), Address varchar (255));
```

★ AND TO INSERT INTO Table:

```
INSERT INTO table-name (colum1 , column 2, ... ) Values ( value 1,
  Value2, .... );
```

★ Extension: To Create a Table using Another Table

Syntax:

```
CREATE TABLE new_table_name AS SELECT column1, column2, ....
 FROM existing_table_name where condition;
```

[This helps us To create a copy of an existing table. The new Table gets The same column Definitions. (NOTE: All columns or specific columns can be selected.)].

SQL DROP TABLE:

The "DROP TABLE" statement is used to drop an existing table in a database.

★ Syntax:

```
DROP Table table_name;
```

SQL TRUNCATE TABLE:

The "TRUNCATE TABLE" statement is used to delete The data inside a table, but not table itself.

★ Syntax:

```
TRUNCATE TABLE table-name;
```

SQL ALTER TABLE:

The "ALTER TABLE" statement is used to add, delete or modify columns in an existing table.

The Alter table statement is also used to add and drop various constraints on an existing table.

★ The Various Alter table Commands Syntax:

- ALTER TABLE - ADD Column: To add a column

Syntax:

```
ALTER TABLE table_name ADD Column_name datatype;
```

- ALTER TABLE - DROP Column: To Delete a column.

Syntax:

```
ALTER TABLE table_name DROP COLUMN Column_name;
```

- ALTER TABLE - RENAME Column: To rename a column:

Syntax:

```
ALTER TABLE table_name RENAME COLUMN old_name TO new_name;
```

- ALTER TABLE - ALTER/ Modify Datatype: To change the datatype of column.

Syntax:

```
ALTER TABLE table_name MODIFY COLUMN Column_name datatype;
```

Or

```
ALTER TABLE table_name MODIFY column_name Datatype;
```

T.T.E...

SQL CONSTRAINTS:

★ Syntax:

```
Create Table Table_name ( Column1 datatype constraints,
                          Column2 datatype constraints,
                          Column3 datatype constraints,
                          ... );
```

The "SQL CONSTRAINTS" are used to specify rules for the data in a table.

CONSTRAINTS are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table.

Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to whole table.

The following are some commonly used SQL CONSTRAINTS:

- NOT NULL → Ensures that a column cannot have a Null value.
- UNIQUE → Ensures that all values in a column are different.
- PRIMARY Key → it is a combination of a NOT NULL AND UNIQUE. Uniquely identifies each row in a table.
- FOREIGN KEY → Prevents actions that would destroy links between tables.
- CHECK → Ensures that a value in a column satisfied a specific condition.
- DEFAULT → Sets a Default value for a column if no value is specified.
- CREATE INDEX → Used to create and retrieve data from the database very quickly.

SQL NOT NULL CONSTRAINTS:

By default a column can hold Null values.

The "NOT NULL" constraint enforces a column to not accept Null values. which means we can't insert a new record (or row), or update a record without adding a value to this field.

★ USING NOT NULL while creating a table:

```
Create table Persons ( Id int NOT NULL , LASTNAME VARCHAR(255
NOT NULL , Firstname varchar(255) NOT NULL , age int );
```

★ SQL NOT NULL ON ALTER TABLE:

```
ALTER TABLE Persons ALTER COLUMN Age int NOT NULL;
```

Or

```
ALTER TABLE Persons Modify column Age int NOT NULL;
```

Or

```
ALTER TABLE Persons Modify Age int NOT NULL;
```

SQL UNIQUE CONSTRAINT:

The "UNIQUE" Constraint ensures that all values in column are different.

Both UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of Columns.

A primary key constraints automaticaly has a unique constraint.

However, we can have many unique constraints per Table, but only one Primary Key constraints per Table.

F.T.E...

USING UNIQUE CONSTRAINT ON CREATE TABLE:

The following SQL creates a UNIQUE constraint on the "ID" column, when the Person Table is created.

Syntax:

```
Create table Persons ( Id int Not NULL UNIQUE, LASTNAME
varchar (255) NOT NULL, FirstNAME varchar (255) );
```

Or

```
Create Table Persons ( Id int NOT NULL, LASTNAME VARCHAR
(255) NOT NULL, FirstName varchar (255), UNIQUE (ID));
```

To APPLY UNIQUE CONSTRAINT ON MULTIPLE COLUMNS:

```
CREATE TABLE Persons ( ID int Not NULL, LASTNAME
Varchar (255) NOT NULL, FirstName Varchar (255),
CONSTRAINT UC_Person UNIQUE ( ID, LASTName));
```

To APPLY UNIQUE CONSTRAINT ON ALTER TABLE:

```
ALTER TABLE PERSON ADD UNIQUE (ID);
```

To APPLY ON MULTIPLE COLUMNS:

```
ALTER TABLE PERSON ADD CONSTRAINTS UC_Person UNIQUE
(ID, LastName);
```

DROP a unique Constraint:

```
ALTER TABLE Person DROP INDEX UC_Person;
```

Or

```
ALTER TABLE Persons DROP CONSTRAINTS UC_Person;
```

SQL PRIMARY KEY CONSTRAINTS:

The "Primary key" constraint uniquely identifies each record in a table.

Primary keys must contain UNIQUE values, and cannot contain Null values.

A table can have only ONE Primary key.

[NOTE: This primary key can consist of single or Multiple columns.]

Syntax:

```
CREATE TABLE Persons ( Id int NOT NULL, Last Name varchar(255)
NOT NULL, Firstname Varchar(255), Age int, Primary Key (ID));
```

Or

```
CREATE TABLE Persons ( Id int NOT NULL PRIMARY KEY, LAST
Name varchar(255) NOT NULL, FirstName varchar(255), Age int);
```

Syntax: To Set Multiple Column as Primary key...

```
CREATE TABLE PERSONS ( Id int NOT NULL, LAST Name varchar(255) NOT NULL
First Name Varchar(255), Age int, CONSTRAINT PK_Person Primary KEY
(Id, LastName));
```

[NOTE: In the example above there is only one Primary Key (PK_Person). However, the value of the Primary key is made up of two Columns (ID + Last Name).]

SQL PRIMARY KEY ON ALTER TABLE:

SYNTAX:

```
ALTER TABLE Persons ADD PRIMARY KEY (ID);
```

DROP a PRIMARY KEY CONSTRAINT:

SYNTAX:

```
ALTER TABLE Persons DROP PRIMARY KEY;
```

or

```
ALTER TABLE Persons DROP CONSTRAINT PK_Person;
```

SQL FOREIGN KEY CONSTRAINTS

The "FOREIGN KEY" Constraints is used To Prevent actions That would destroy link between tables.

A FOREIGN KEY is a field (or collection of fields) in one table, That refers To Primary Key in Another Table.

NOTES: The table with foreign Key is called a child table, and The Table That refers To Primary Key (or with Primary Key) is called Parent Table.

The "foreign Key" constraint Prevents invalid data from being inserted into The foreign Key column, because it has to be One of The values contained in The Parent Table.

★ Syntax:

```
CREATE TABLE Orders ( orderid int NOT NULL, OrderNumber int NOT NULL, PersonId int, Primary Key (OrderId), Foreign Key (PersonId) Refrences Persons (PersonId));
```

Or

```
CREATE TABLE Orders ( Orderid int NOT NULL PRIMARY KEY, OrderNUMBER int NOT NULL, PersonId int FOREIGN KEY Refrences Persons (PersonId));
```

SQL FOREIGN KEY ON ALTER TABLE:

To create a "FOREIGN KEY" constraint on The "PersonId" column when The "Orders" table is already created.

Syntax:

```
ALTER TABLE Orders Add FOREIGN KEY (PersonId) Refrences Persons (PersonId);
```

DROP a FOREIGN KEY CONSTRAINT:

Syntax:

```
ALTER TABLE Orders DROP FOREIGN KEY FK_PersonOrder;
```

Or

```
ALTER TABLE Orders DROP CONSTRAINT FK_PersonOrder;
```

SQL CHECK CONSTRAINT:

The "CHECK" constraint is used to limit the value range that can be placed in a column.

if you define a check constraint on a column it will allow only certain values for this column.

if you define a check constraint on a table it can limit the values in certain columns based on values in other columns in the row.

★ Syntax:

```
CREATE TABLE Persons ( Id int NOT NULL, LastName varchar (255)
NOT NULL, FirstName varchar (255), Age int, CHECK (Age >= 18));
```

★ CHECK CONSTRAINT ON MULTIPLE Column:

```
CREATE TABLE Persons ( Id int NOT NULL, LastName varchar (255)
NOT NULL, FirstName varchar (255), Age int, city varchar (255),
CONSTRAINT CHK_Person CHECK (Age >= 18 AND CITY = "sandes"))
```

SQL CHECK ON ALTER TABLE:

```
ALTER TABLE Persons ADD CHECK (Age >= 18);
```

SQL DEFAULT CONSTRAINT:

The "DEFAULT" CONSTRAINT is used to set a default value for a column.

The default value will be added to all new records, if no other value is specified.

★ Syntax:

```
CREATE TABLE Persons ( Id int NOT NULL, LastName varchar (255)
NOT NULL, Firstname varchar (255), Age int, City varchar (255)
DEFAULT 'sandnes');
```

Or

```
CREATE TABLE Orders ( Id int NOT NULL, OrderNumber int NOT NULL,
Order-date date Default Getdate());
```

SQL AUTO INCREMENT FIELD:

Auto-increment allows a unique number to be generated automatically when a new record is inserted into a Table.

Often this is the primary key field that we would like to be created automatically every time a new record is inserted.

★ Syntax:

```
CREATE TABLE Persons ( Personid int NOT NULL AUTO_INCREMENT,
LastName Varchar (255) NOT NULL , FirstName Varchar (255), Age int,
PRIMARY KEY (Personid));
```

[NOTE: By default, the starting value for Auto_INCREMENT is 1, and it will increment by 1 for each new record. To let Auto-increment sequence start with Another value.]

★ Syntax:

```
ALTER TABLE Persons AUTO_INCREMENT = 100;
```

SQL Date Data Types:

- DATE : format yyyy-MM-DD
- DATETIME: format yyyy-MM-DD HH:MI:SS
- TIMESTAMP: format yyyy-MM-DD HH:MI:SS
- YEAR : format yyyy or yy

eg: SELECT * FROM Orders WHERE OrderDate = '2008-11-11'

P.T.O...

SQL DATA TYPES FOR MySQL, SQL SERVER & MS ACCESS:

★ STRING DATA TYPES:

DATA TYPE	DESCRYPTION
• CHAR (size)	A Fixed Length string (can contain letters, number and special characters). The size Parameter specifies column length in characters - can be from 0 to 255. Default is 1
• Varchar (size)	(Can contain letter, number & special characters) Max length → 0 to 65535.
• BINARY (size)	Equal to CHAR(), But stores binary byte string.
• VARBINARY (size)	used To store Binary Byte strings.
• TINYBLOB	for BLOBs (Binary Large Objects). Max Length: 255 Bytes
• ENUM (val1, val2..)	A string object That can have only one value, Choosen from a list of Possible values. if a value is inserted that is not in the list, a blank value will be inserted.
• SET (val1, val2...)	A string object That can have 0 or more values, chosen from a list of Possible values

★ Numeric Data Types:

DATA TYPE	DESCRYPTION
• INT (size)	used To store INTEGER VALUE.
• Float (size, d)	The total number of digits is specified in size. The number of digits after The decimal Point is specified in The d Parameter.