Topics covered

1) SQL JOINS
   (INNER JOIN, LEFT JOIN, RIGHT JOIN, SELF JOIN)
2) SQL UNION
3) UNION ALL
4) GROUP BY
5) HAVING
6) INSERT INTO
7) CREATE TABLE using another table
8) CONSTRAINTS
   (NOTNULL, UNIQUE, PRIMARY KEY, FOREIGN KEY,
   CHECK, DEFAULT, INDEX)
9) AUTO_INCREMENT

SQL Joins:

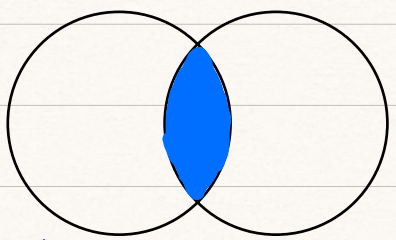
A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Different Types of SQL JOIN's:

- INNER JOIN
- Left JOIN
- Right JOIN
- SELF JOIN
- FULL OUTER JOIN (not supported in MY SQL, instead we can use Left JOIN & Right JOIN)

INNER JOIN:

Returns records that have matching values in both tables.



Syntax:

```
SELECT column1, column2,....
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

Example:

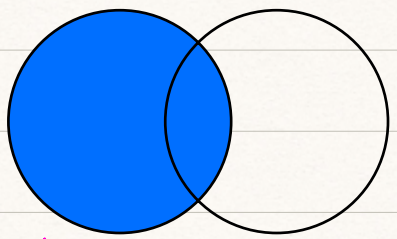
```
SELECT orders.orderID,
       customer.Customer_name
FROM  orders
INNER JOIN Customers
 ON orders.customerID = Customers.customerID;
```

```
SELECT O.orderID, C.Customer_name,
       S.shipperName
FROM  orders AS O, Customers AS C,
       Shippers AS S
INNER JOIN Customers
       ON O.customerID = C.customerID
INNER JOIN Shippers
       ON O.shipperID = S.ShipperID;
```

* We can join n number of tables but the condition is that both the tables share column with same data.

LEFT JOIN:

The LEFT JOIN returns all the records from the Left table (table1), and matched records from the right table (table 2). The result is NULL from the right side, if there is no match.



* Sometimes LEFT JOIN is called LEFT OUTER JOIN.

Syntax:

```
SELECT column1, column2,....
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

Example:

```
SELECT orders.orderID,
       customer.Customer_name
FROM orders
LEFT JOIN Customers
```

```
ON Orders.CustomerID = Customers.CustomerID
ORDER BY Customers.Customer_name;
```

RIGHT JOIN:

The RIGHT JOIN returns all records from the right table (table2), and the matched records from the left table (table1). The result is NULL from the left side, when there is no match.

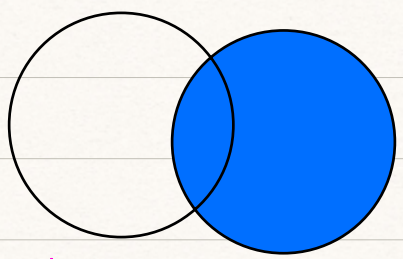


table1 table2

* Sometimes RIGHT JOIN is called RIGHT OUTER JOIN.

Syntax:

```
SELECT column1, column2,....
FROM table1
RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```

Example:

```
SELECT O.orderID, E.LastName,
```

```
E.FirstName
FROM orders AS O, Employees AS E
RIGHT JOIN Employees
ON O.EmployeeID = E.Employee ID
ORDER BY Orders.Orders9D;
```

FULL OUTER JOIN : (NOT Available in MYSQL instead we use LEFT OUTER JOIN & RIGHT OUTER JOIN)

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

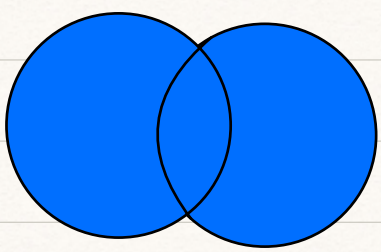


table1 table2

Syntax:

```
SELECT column1, column2,....
FROM table1
FULL OUTER JOIN table2
ON table1.column_name = table2.column_name;
```

* FULL OUTER JOIN Can potentially result large result-sets.

SELF JOIN:

A SELF JOIN is a regular join, but a table is joined with itself.

Syntax:

```
SELECT column1, column2, ....
FROM table1 AS T1, table1 AS T2
WHERE condition;
```

Example:

```
SELECT A.Customer_name AS CustomerName1,
    B.Customer_name AS CustomerName2,
    A.City
FROM Customers AS A, Customers AS B
WHERE A.CustomerID <> B.CustomerID
    AND A.City = B.City
ORDER BY A.City;
```

* In MySQL the default JOIN is INNER JOIN.

SQL UNION:

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

* Each SELECT statement within UNION must have the same number of columns.
* The columns must also have simillar data types.
* The columns in each SELECT statement must also be in the same order.

UNION Syntax:

```
SELECT Column1, column2, ....
FROM table1
UNION
SELECT column1, Column2, ....
FROM table2;
```

* UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL:

UNION ALL Syntax:

```
SELECT column1, column2, ....
FROM table1
UNION ALL
SELECT column1, column2, ....
FROM table2;
```

* The column names in the result-set are usually equal to the column names in the first SELECT statement in the UNION.

Example:

```
SELECT City
FROM Customers
UNION
SELECT City
FROM Suppliers;
```

```
SELECT City
FROM Customers
UNION ALL
SELECT City
```

UNION ALL with ORDER BY

```
FROM Suppliers
ORDER BY City;
```

UNION with WHERE

```
SELECT City
FROM Customers
WHERE Country = 'India'
UNION ALL
SELECT City
FROM Suppliers
WHERE Country = 'India'
ORDER BY City;
```

UNION ALL with WHERE

```
SELECT City, Country
FROM Customers
WHERE Country = 'India'
UNION ALL
SELECT City, Country
FROM Suppliers
WHERE Country = 'India'
ORDER BY City;
```

GROUP BY:

The GROUP BY clause is used to group rows based on one or more columns in a table. It is often used in combination with aggregate functions, such as COUNT(), SUM(), AVG(), etc. to perform calculations on groups of rows

Syntax:

```
SELECT column1, column2,...
FROM <table-name>
WHERE condition
GROUP BY column1, column2,...
ORDER BY column1, column2,...
```

(or)

```
SELECT column1, aggregate_function(column2)
FROM <table_name>
GROUP BY column1;
```

* The GROUP BY clause groups the result-set by the specified columns. Each unique value in the columns forms a group, and the aggregate function(s) are applied to each group separately.

Example:

```
SELECT category,
       SUM(quantity) As total_quantity
FROM Sales
GROUP BY category;

SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country;

SELECT COUNT (CustomerID), Country
FROM Customers
GROUP BY Country;
ORDER BY COUNT (CustomerID) DESC;

SELECT S.ShippersName,
       COUNT(O.OrdersID) As NumberOfOrders
FROM ORDERS AS O
LEFT JOIN shippers AS S
ON O.shipperID = S.ShipperID
GROUP BY ShipperName;
```

* Columns in the SELECT clause must either be part of the GROUP BY clause or be used with aggregate functions. This ensures that non-aggregated columns in the SELECT clause correspond to the grouped rows.
* The GROUP BY clause can include multiple columns, seperated by commas. In that case, the result set will be grouped based on the unique combinations of the specified columns.

SQL HAVING:

The Having clause is used to filter the results of a query based on conditions applied to aggregated values. It is often used in combination with the GROUP BY clause to specify conditions on groups of rows.

* WHERE cannot be used on conditions applied to aggregated values.
* HAVING clause is used for aggregated values
* HAVING clause is also used on the result-set after grouping.

Syntax:

```
SELECT column1, column2,...
FROM <table_name>
WHERE condition
GROUP BY column1, column2,...
HAVING condition
ORDER BY column1, column2,...
```

Example:

```
SELECT COUNT(customerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT (CustomerID) > 5;
```

```
SELECT COUNT(customerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT (CustomerID) > 5
ORDER BY COUNT(CustomerID) DESC;
```

```
SELECT E.LastName,
       COUNT(O.ordersID) AS NumberofOrders
FROM (Orders AS O
INNER JOIN EMPLOYEES AS E
ON O.EmployeeID = E.EmployeeID)
GROUP BY LastName
HAVING COUNT(O.ordersID) >10;
```

```
SELECT E.LastName,
       COUNT(O.ordersID) AS NumberofOrders
```

```
FROM (Orders AS O
INNER JOIN EMPLOYEES AS E
ON O.EmployeeID = E.EmployeeID)
WHERE LastName = 'xxx' OR
      LastName = 'yyy'
GROUP BY LastName
HAVING COUNT(O.ordersID) > 25;
```

SQL INSERT INTO:

The INSERT INTO SELECT statement copies data from one table and inserts it into another table.

* INSERT INTO SELECT requires that data types in source and target tables match
* The existing records in the target table are unaffected.

Syntax:

Copy all columns from one table to another table:

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

Copy only some columns from one table into another table:

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

Example:

```
INSERT INTO Customers (
        customername, city, country)
SELECT SupplierName, City, Country
FROM Suppliers;
```

```
INSERT INTO Customers (
        CustomerName, ContactName, Address,
        City, PostalCode, Country)
SELECT SupplierName, ContactName, Address,
        City, PostalCode, Country
FROM Suppliers;
```

```
INSERT INTO Customers (
        CustomerName, City, Country)
SELECT SupplierName, City, Country
FROM Suppliers
WHERE Country = 'India';
```

Create Table using Another Table:

A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement.

* The new table gets the same column definitions. All columns or specific columns can be selected.
* If you create a new table using an existing table, the new table will be filled with the existing values from the old table.

Syntax:

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

SQL Constraints:

SQL constraints are used to specify rules for data in a table.

SQL Create Constraints

Constraints can be specified when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement.

Syntax:

```
CREATE TABLE table_name(
        Column1 datatype constraint,
        Column2 datatype constraint,
        column3 datatype constraint,
        ....
       );
```

SQL Constraints:

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. If there is any violation between the constraint and the data action, the action is aborted.

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

The following constraints are commonly used in SQL:

* **NOT NULL** - Ensures that a column have a NULL value.
* **UNIQUE** - Ensures that all values in a column are different.
* **PRIMARY KEY** - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table.
* **FOREIGN KEY** - Uniquely identifies a row/record in another table
* **CHECK** - Ensures that all values in a column when no value is specified.
* **DEFAULT** - sets a default value for a

Column when no value is specified

*INDEX - Use to create and retreive data from the database very quickly.

NOT NULL Constraint:

By default a column can hold NULL values. The NOT NULL Constraint enforces a column to NOT accept NULL values.

* This enforces a field to always contain a value, which means that we cannot insert a new record, or update a record without adding a value to this field.

Example: The following SQL ensures that the "ID", "LastName", and "FirstName" columns will NOT accept NULL values:

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName Varchar(255) NOT NULL,
    FirstName Varchar(255) NOT NULL,
    Age int
    );
```

* If the table has already been created, you can add a NOT NULL constraint to a column with the ALTER TABLE statement.

UNIQUE Constraint:

The UNIQUE constraint ensures that all values in a column are different.

* Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or a set of columns.
* A PRIMARY KEY constraint automatically has a UNIQUE constraint.
* We can have many UNIQUE constraints in a table but only one PRIMARY KEY constraint.

UNIQUE Constraint on CREATE TABLE

The following SQL creates a UNIQUE Constraint on the "ID" column when the "Persons" table is created.

Syntax:

```
CREATE TABLE Persons (
    ID int NOT NULL UNIQUE,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int
    );
```

UNIQUE on multiple columns:

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CONSTRAINT UN_Person UNIQUE(ID,
                                LastName)
    );
```

UNIQUE Constraint on ALTER TABLE:

* To create UNIQUE constraint on TABLE already created.

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

(or)

* If we want to add the constraint on multiple columns

```
ALTER TABLE Persons
ADD CONSTRAINT UN_Person
    UNIQUE (ID, LastName);
```

DROP a UNIQUE Constraint

```
ALTER TABLE Persons
DROP CONSTRAINT UN_Person;
```

PRIMARY KEY Constraint :

The primary key constraint uniquely identifies each record in a database table.

* Primary key must contain UNIQUE values and cannot contain NULL values.
* A table can only have one primary key, which may consist of single or multiple fields.

PRIMARY KEY on CREATE TABLE :

```
CREATE TABLE Persons (
    ID int NOT NULL PRIMARY KEY,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int
);
```

PRIMARY KEY naming and on Multiple Columns:

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
```

```
    Age int,
    CONSTRAINT PK_Person
        PRIMARY KEY (ID, LastName)
);
```

* In the above example there is only one PRIMARY KEY (PK_Person). However, the value of the primary key is made up of Two columns (ID + LastName).

PRIMARY KEY on ALTER TABLE:

* To create PRIMARY KEY constraint on TABLE already created.

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

(or)

* If we want to add the constraint on multiple columns

```
ALTER TABLE Persons
ADD CONSTRAINT PK_Person
 PRIMARY KEY (ID, LastName);
```

* Note: If we use ALTER TABLE to add a PRIMARY KEY, the PRIMARY KEY column(s) must already have been declared to not contain NULL values (when the table was first created).

DROP a PRIMARY KEY Constraint:

```
ALTER TABLE Persons
DROP CONSTRAINT PK_Persons;
```

FOREIGN KEY Constraint:

A FOREIGN KEY constraint is used to link two tables together.

* A FOREIGN KEY in a table points to a PRIMARY KEY in another table.
* First, specify the name of FOREIGN KEY Constraint that we want to create after the CONSTRAINT key word. If we omit the constraint name, MySQL automatically generates a name for the FOREIGN KEY constraint.
* Second, specify a list of comma-seperated foreign key columns after the FOREIGN KEY keywords. The foreign key name is also optional and is generated automatically if you skip it.
* Third, specifically the parent table followed by a list of comma-seperated columns to which the FOREIGN KEY columns reference.
* Finally, specify how foreign key maintains the referential integrity between the child and parent tables by using the ON DELETE and

ON UPDATE clauses.

* The reference option determines action which MYSQL will take when values in the parent key columns are deleted (ON DELETE) or updated (ON UPDATE).

Syntax:

```
CREATE TABLE orders(
    OrderID int NOT NULL PRIMARY KEY,
    OrderNumber int Not NULL,
    personID int FOREIGN KEY
    REFERENCES persons(personID)
);
```

MYSQL has four reference options:

1) CASCADE
2) SET NULL
3) NO ACTION
4) RESTRICT

CASCADE: If a row from the parent table is deleted or updated. the values of the matching rows in the child table automa-

tically deleted or updated.

SETNULL: If a row from the parent table is deleted or updated, the values of the foreign key column (or columns) in the child table are set to NULL.

RESTRICT: If a row from the parent table has a matching row in the child table, MYSQL rejects deleting or updating rows in the parent table.

NO ACTION: It is same as RESTRICT.

* MY SQL fully supports all three actions: RESTRICT, CASCADE and SETNULL.

** If we don't specify the ON DELETE and ON UPDATE clause, the default action is RESTRICT.

```
CREATE TABLE orders(
    OrderID int NOT NULL PRIMARY KEY,
    OrderNumber int NOT NULL,
    personID int FOREIGN KEY
```

```
    REFERENCES  persons(personID)
        ON UPDATE  SET NULL
        ON DELETE  SET NULL
);
```

Disabling FOREIGN KEY checks:

Sometimes it is very useful to disable foreign key checks. e.g., when we import data from a csv file in to a table. If we don't disable foreign key checks, we have the data in the exact order (i.e parent tables first and then child tables), this is tedious. However, if we disable foreign key checks we can then load in any order.

FOREIGN KEY on ALTER TABLE:

To create FOREIGN KEY RESTRAINT when table is already created.

```
ALTER TABLE orders
ADD FOREIGN KEY (personID)
    REFERENCES persons(personID);
```

Defining the CONSTRAINT on multiple columns

```
ALTER TABLE orders
ADD FK_person
    FOREIGN KEY (personID)
    REFERENCES persons (PersonID);
    ON UPDATE CASCADE
    ON DELETE CASCADE;
```

DROP a FOREIGN KEY Constraint:

```
ALTER TABLE orders
DROP CONSTRAINT FK_Person;
```

CHECK CONSTRAINT:

If we specify a column with CHECK CONSTRAINT, values can only be inserted if the value satisfies a specific condition that is mentioned.

CHECK on CREATE TABLE:

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

* Here it will only accept data into Age column where Age is greater than or equal to 18.

CHECK on multiple columns:

```
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='Hyd')
);
```

CHECK on ALTER TABLE:

To create a CHECK after the table is created.

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

CHECK on Multiple columns in ALTER TABLE:

```
ALTER TABLE Persons
ADD CONSTRAINT chk_AgeCity
  CHECK (Age>=18 AND City='Hyd');
```

DROP a CHECK CONSTRAINT:

```
ALTER TABLE Persons
DROP CONSTRAINT chk_AgeCity;
```

DEFAULT CONSTRAINT:

The DEFAULT CONSTRAINT is used to provide 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 'Hyd'
);
```

* If no value is mentioned for city in data the default value of 'Hyd' will be taken.

The DEFAULT CONSTRAINT can also be used to insert system values, by using function like GETDATE():

```
CREATE TABLE Orders(
    ID int NOT NULL,
    OrderNumber int NOT NULL,
    OrderDate date DEFAULT GETDATE()
```

```
);
```

DEFAULT on ALTER TABLE:

```
ALTER TABLE Persons
ALTER COLUMN City SET DEFAULT 'Hyd';
```

DROP A DEFAULT Constraint:

```
ALTER TABLE Persons
ALTER COLUMN City DROP DEFAULT;
```

CREATE INDEX statement:

The CREATE INDEX Statement is used to create indexes in tables.

* Indexes are used to retreive data from the database very fast.
* The users cannot see the indexes, they are just used to speed up searches/queries.

* Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched against.

CREATE INDEX syntax:

```
CREATE INDEX index_name
ON <table_name> (column1, column2,.... );
```

* Creates an index on a table. Duplicate values are allowed.

CREATE UNIQUE INDEX syntax:

```
CREATE UNIQUE INDEX index_name
ON <table_name> (column1, column2,....);
```

* Creates a unique index on a table. Duplicate values are not allowed.

* The syntax of creating indexes varies based on database.

Example:

```
CREATE INDEX idx_lastname
ON Persons (LastName);
```

* Creates an index named 'idx_lastname' on the LastName column in the Persons table.

```
CREATE INDEX idx_pname
ON Persons (LastName, FirstName)
```

* we can create index on combination of columns. we have to list the column names within paranthesis, seperated by commas.

* Indexing on multiple columns can be done when the columns are frequently used together as filters. A multi-column index performs better than multiple single column indexes.

```
CREATE UNIQUE INDEX idx_emailId
ON users(email);
```

* creates unique index for every email id in users table.

* sometimes we only want to index first N characters of a string.

```
CREATE INDEX idx_CompanyPartName
ON companies(Name(20));
```

* In MYSQL 8.x we can also specify the storage order of a column in an index. This can be helpful if we need to display the column in a certain order. By default it is ascending order:

```
CREATE INDEX idx_reverseName
ON companies (Name DESC);
```

* In MySQL 8.0-13 and above it supports functional key parts. These index expression values rather than column or column prefix values.
* They enable indexing of values not stored directly in the table.
* we need to use double paranthesis

```
CREATE INDEX idx_totalSales
ON products ((product_price*quantity));
```

DROP INDEX Syntax:

To Drop a non-primary key index, we use

```
DROP INDEX index_name
ON <table_name>
```

* The syntax requires the table name to be specified because MYSQL allows index names to be reused on multiple tables.
* Primary keys in MYSQL are always named PRIMARY (not case sensitive). But because

PRIMARY is a reserved keyword, quotes are required when we use it in DROP INDEX command.

```
DROP INDEX 'PRIMARY'
ON <table_name>;
```

→ here it is not keyword by just name

ALTER TABLE Syntax to DROP INDEX:

```
ALTER TABLE <table_name>
DROP INDEX index_name;
```

Drop the PRIMARY key.

```
ALTER TABLE <table_name>
DROP PRIMARY KEY;
```

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 (
    ID int PRIMARY KEY AUTO_INCREMENT,
    LastName Varchar(255) NOT NULL,
    FirstName Varchar(255),
    Age int
);
```

* MySQL uses the AUTO_INCREMENT keyword to perform an auto-increment feature
* By default, the starting value for AUTO-INCREMENT is 1, and it will increment by 1 for each new record.
* TO let the AUTO-INCREMENT sequence start with another value we can use ALTER TABLE

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

* To insert a new record in to the "Persons" table, we will NOT have to specify a value for the 'ID' column (a unique value will be added automatically).

```
INSERT INTO Persons (FirstName, LastName)
VALUES ('shuba', 'Bhupathiraju');
```

* The SQL statement would insert a new record in to the "Persons" table. The "ID" column would be assigned a unique value

** This Syntax is only applicable to MYSQL. It is different for other databases.