MySQL Query

- To create a Data Base

  Create Database “Database_name”;

- To drop a Data Base

  Drop Database “Database_name”;

- To crate a Table

  Create Table “Table_name”
  (“column name1” Data type [argument] ,
  “column name2” Data type [argument] ,
  …….);

  Arguments
  NOT NULL: - ensure that the column cannot have null value
  DEFAULT: - provide a default value when none is specified
  UNIQUE: - ensure that all values in a column are different
  CHECK: - make sure that all values in column satisfy the condition
  Primary key: - used to uniquely identify a row in the table.
  Foreign Key: - used to ensure referential integrity of the data.

  Data type
  INT, CHAR, VARCHAR, BIGINT, DATE, BOOL, TIMESTAMP(0)

- To drop the table

  Drop Table “Table_name”;

- To insert rows into table

  - The INSERT INTO statement is used to add new records into a database table

  Syntax

  INSERT INTO "table_name" ("column1", "column2", ...)

  VALUES ("value1", "value2", ...);

  - Example
    - Single row (without column names specified)

      INSERT INTO customer_table VALUES
      (1, ‘bee', ’cee’, 32, ‘bc@xyz.com’);

- Single row (with column names specified)

```
INSERT INTO customer_table ( cust_id, first_name, age, email_id)
VALUES
        (2, 'dee', 22, 'd@xyz.com');
```

- Multiple rows

```
INSERT INTO customer_table VALUES
    (1, 'ee', 'ef', 35, 'ef@xyz.com' ),
    (1, 'gee', 'eh', 42, 'gh@xyz.com' ),
    (1, 'eye', 'jay', 62, 'ij@xyz.com' ));
```

➢ To delete the content of table (rows)

```
Delete from "Table_name";
```

➢ To import data from csv or txt file

- The basic syntax to import data from CSV file into a table using COPY statement is as below

Syntax

```
COPY "table_name" ("column1", "column2", ...) FROM
'C:\tmp\persons.csv' DELIMITER ',' CSV HEADER;

From text file
COPY "table_name" ("column1", "column2", ...) FROM
'C:\tmp\persons.txt' DELIMITER ',';
```

➢ To select the data from table

- The SELECT statement is used to fetch the data from a database table which returns this data in the form of a result table. These result tables are called result-sets.

Syntax

1. for 1 or 2 column

```
SELECT "column_name1", "column_name2", "column_name3"
    FROM "table_name";
```

2. for all columns

```
SELECT * FROM "table_name";
```

- Example

The SELECT statement is used to fetch the data from a database table

1) Select one column

```
SELECT first_name FROM customer_table;
```

2) Select multiple columns

```
SELECT first_name, last_name FROM customer_table;
```

3) Select all columns

```
SELECT * FROM customer_table;
```

➢ To select distinct data from table

- The DISTINCT keyword is used in conjunction with the SELECT statement to eliminate all the duplicate records and fetching only unique records.

  Syntax

```
SELECT DISTINCT "column_name"
    FROM "table_name";
```

- Example
  - Select one column

```
SELECT DISTINCT customer_name
    FROM customer_table;
```

  - Select multiple columns

```
SELECT DISTINCT customer_name, age
    FROM customer_table;
```

➢ Where statement

- The SQL WHERE clause is used to specify a condition while fetching the data from a single table or by joining with multiple tables. If the given condition is satisfied, then only it returns a specific value from the table.

  Syntax

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

- Example :
  - Equals to condition

```
SELECT first_name FROM customer_table WHERE age = 25;
```

  - Less than/ Greater than condition

```
SELECT first_name, age FROM customer_table WHERE age>25;
```

- Matching text condition

  SELECT * FROM customer_table WHERE first_name = “John”;

➢ AND or OR statement

- The SQL AND & OR operators are used to combine multiple conditions to narrow data in an SQL statement. These two operators are called as the conjunctive operators.

  Syntax

  SELECT "column_name"
  FROM "table_name"
  WHERE "simple condition"
  { [AND|OR] "simple condition"}+;

- Examples:
  - SELECT first_name, last_name, age
    FROM customer_table
    WHERE age > 20 AND age < 30 ;

  - SELECT first_name, last_name, age
    FROM customer_table
    WHERE age < 20 OR age >30 OR first_name = ‘John’;

➢ Not Statement :

- NOT condition is used to negate a condition in a SELECT, INSERT, UPDATE, or DELETE statement.

  Syntax

  SELECT "column_name"
  FROM "table_name"
  WHERE NOT "simple condition";

- Example:
  - SELECT first_name,last_name, age
    FROM employee
    WHERE NOT age=25 ;

  - SELECT first_name,last_name, age
    FROM employee
    WHERE NOT age=25 AND NOT first_name = ‘JAY’;

➢ Update statement

- The SQL UPDATE Query is used to modify the existing records in a table.

  Syntax

```
UPDATE "table_name"
SET column_1 = [value1], column_2 = [value2], ...
WHERE "condition";
```

- Examples:
  - Single row (with column names specified)

```
UPDATE Customer_table
SET Age = 17, Last_name = 'Pe'
WHERE Cust_id = 2;
```

  - Multiple rows

```
UPDATE Customer_table
SET email_id = 'gee@xyz.com
WHERE First_name = 'Gee' or First_name = 'gee';
```

➢ Delete statement :

- The DELETE Query is used to delete the existing records from a table.

  Syntax

```
DELETE FROM "table_name"
WHERE "condition";
```

- Example :
  - Single row

```
DELETE FROM CUSTOMERS
WHERE ID = 6;
```

  - Multiple rows

```
DELETE FROM CUSTOMERS
WHERE age>25 ;
```

  - All rows

```
DELETE FROM CUSTOMERS;
```

➢ Alter Statement :

- The ALTER TABLE statement is used to change the definition or structure of an existing table

Syntax

```
ALTER TABLE "table_name" [Specify Actions];
```

Following actions can be performed

- Columns – Add, Delete (Drop), Modify or Rename
- Constraints – Add, Drop
- Index – Add, Drop

➢ Column add or Drop

- The basic syntax of an ALTER TABLE command to add/drop a Column in an existing table is as follows.

Syntax

```
ALTER TABLE "table_name"
ADD "column_name" "Data Type";

ALTER TABLE "table_name"
DROP "column_name";
```

➢ Column Rename / Modify

- The basic syntax of an ALTER TABLE command to Modify/Rename a Column in an existing table is as follows.

Syntax

```
ALTER TABLE "table_name"
ALTER COLUMN "column_name"
TYPE "New Data Type";

ALTER TABLE "table_name"
RENAME COLUMN "column 1" TO "column 2";
```

➢ Constraint add or drop

- The basic syntax of an ALTER TABLE command to add/drop a Constraint on a existing table is as follows.
  - Syntax 1.

```
ALTER TABLE "table_name“
ALTER COLUMN “column_name”
SET NOT NULL;
```

- 2.

```
ALTER TABLE "table_name“
ALTER COLUMN “column_name”
 DROP NOT NULL;
```

- 3.

```
ALTER TABLE "table_name"
ADD CONSTRAINT “column_name”
CHECK (“column_name”>=100);
```

- 4.

```
ALTER TABLE "table_name"
ADD PRIMARY KEY (“column_name”);
```

- 5.

```
ALTER TABLE “child_table"
ADD CONSTRAINT “child_column”
FOREIGN KEY (“parent column”) REFERENCES “parent table”;
```

➢ IN statement

- IN condition is used to help reduce the need to use multiple OR conditions in a SELECT, INSERT, UPDATE, or DELETE statement.

  Syntax

```
SELECT "column_name"
FROM "table_name"
WHERE "column_name" IN ('value1', 'value2', ...);
```

- Example:
  - SELECT * FROM customer
    WHERE city IN (‘Philadelphia’, ‘Seattle’)
  - SELECT * FROM customer
    WHERE city = ‘Philadelphia’ OR city = ’Seattle’;

➢ Between Statement

- The BETWEEN condition is used to retrieve values within a range in a SELECT, INSERT, UPDATE, or DELETE statement.

Syntax

```
SELECT "column_name"
FROM "table_name"
 WHERE "column_name" BETWEEN 'value1' AND 'value2';
```

- Example:
  - SELECT * FROM customer
    WHERE age BETWEEN 20 AND 30;

    Which is same as

    ```
    SELECT * FROM customer
    WHERE age>= 20 AND age<= 30;
    ```

  - SELECT * FROM customer
    WHERE age NOT BETWEEN 20 and 30;

  - SELECT * FROM sales
    WHERE ship_date BETWEEN '2015-04-01' AND '2016-04-01';

➢ Like Statement

- The PostgreSQL LIKE condition allows you to perform pattern matching using Wildcards.

  Syntax

  ```
  SELECT "column_name"
  FROM "table_name"
  WHERE "column_name"
  LIKE {PATTERN};
  ```

  {PATTERN} often consists of wildcards

- Example:
  - Wildcard Explanation

    % Allows you to match any string of any length (including zero length)

    _ Allows you to match on a single character

    A% means starts with A like ABC or ABCDE
    %A means anything that ends with A
    A%B means starts with A but ends with B
    AB_C means string starts with AB, then there is one character, then there is C

- SELECT * FROM customer_table
  WHERE first_name LIKE 'Jo%';

- SELECT * FROM customer_table
  WHERE first_name LIKE '%od%';

- SELECT first_name, last_name
  FROM customer_table
  WHERE first_name LIKE 'Jas_n';

- SELECT first_name, last_name
  FROM customer_table
  WHERE last_name NOT LIKE 'J%';

- SELECT * FROM customer_table
  WHERE last_name LIKE 'G\%';

➢ Order BY function

- The ORDER BY clause is used to sort the records in result set. It can only be used in SELECT statements.

  Syntax

  ```
  SELECT "column_name"
  FROM "table_name" [WHERE "condition"]
  ORDER BY "column_name" [ASC, DESC];
  ```

- It is possible to order by more than one column.

  ```
  ORDER BY "column_name1" [ASC, DESC],
           "column_name2" [ASC, DESC];
  ```

- Example:
  - SELECT * FROM customer
    WHERE state = 'California'
    ORDER BY Customer_name;

    Same as
    SELECT * FROM customer
    WHERE state = 'California'
    ORDER BY Customer_name ASC;

- SELECT * FROM customer
  ORDER BY 2 DESC;

- SELECT * FROM customer
  WHERE age>25
  ORDER BY City ASC, Customer_name DESC;

- SELECT * FROM customer
  ORDER BY age;

➢ Limit statement

- LIMIT statement is used to limit the number of records returned based on a limit value.

  Syntax

  ```
  SELECT “column_names”
  FROM “table_name” [WHERE conditions]
  [ORDER BY expression [ ASC | DESC ]
  LIMIT row_count;
  ```

- Example:
  - SELECT * FROM customer
    WHERE age >= 25
    ORDER BY age DESC
    LIMIT 8;

  - SELECT * FROM customer
    WHERE age >=25
    ORDER BY age ASC
    LIMIT 10;

➢ AS statement

- The keyword AS is used to assign an alias to the column or a table. It is inserted between the column name and the column alias or between the table name and the table alias.

  Syntax

  ```
  SELECT column_name" AS "column_alias"
  FROM "table_name“;
  ```

- Example
  - SELECT Cust_id AS “Serial number”,
    Customer_name as name,
    Age as Customer_age
    FROM Customer ;

➢ Aggregate function COUNT:

- Count function returns the count of an expression

  Syntax

  SELECT "column_name1",
  COUNT ("column_name2") FROM "table_name”;

- Example:
  - SELECT COUNT(*)
    FROM sales;

  - SELECT COUNT (order_line) as "Number of Products Ordered",
    COUNT (DISTINCT order_id) AS "Number of Orders"
    FROM sales WHERE customer_id = 'CG-12520';

➢ Aggregate Function SUM:

- Sum function returns the summed value of an expression

  Syntax

  SELECT sum(aggregate_expression)
  FROM tables [WHERE conditions];

- Example
  - SELECT sum(Profit) AS "Total Profit"
    FROM sales;

  - SELECT sum(quantity) AS “Total Quantity"
    FROM orders where product_id = ‘FUR-TA-10000577’;

➢ Aggregate Function AVG:

- AVG function returns the average value of an expression.

  Syntax

  SELECT avg(aggregate_expression)
  FROM tables [WHERE conditions];

- Example:
  - SELECT avg(age) AS "Average Customer Age"
    FROM customer;

- SELECT avg(sales * 0.10) AS "Average Commission Value"
  FROM sales;

➢ Aggregate Function MIN/MAX:

- MIN/MAX function returns the minimum/maximum value of an expression.

  Syntax

  ```
  SELECT min(aggregate_expression)
  FROM tables [WHERE conditions];

  SELECT max(aggregate_expression)
  FROM tables [WHERE conditions];
  ```

- Example:
  - SELECT MIN(sales) AS Min_sales_June15
    FROM sales
    WHERE order_date BETWEEN '2015-06-01' AND '2015-06-30';
  - SELECT MAX(sales) AS Max_sales_June15
    FROM sales
    WHERE order_date BETWEEN '2015-06-01' AND '2015-06-30';

➢ GROUP BY statement:

- GROUP BY clause is used in a SELECT statement to group the results by one or more columns.

  Syntax

  ```
  SELECT "column_name1", "function type" ("column_name2")
  FROM "table_name"
  GROUP BY "column_name1"
  ```

- Example:
  - SELECT region, COUNT (customer_id) AS customer_count
    FROM customer GROUP BY region;
  - SELECT product_id, SUM (quantity) AS quantity_sold
    FROM sales
    GROUP BY product_id ORDER BY quantity_sold DESC;

- SELECT customer_id, MIN(sales) AS min_sales,
  MAX(sales) AS max_sales,
  AVG(sales) AS Average_sales,
  SUM(sales) AS Total_sales
  FROM sales GROUP BY customer_id
  ORDER BY total_sales DESC LIMIT 5;

➢ HAVING Statement:

- HAVING clause is used in combination with the GROUP BY clause to restrict the groups of returned rows to only those whose the condition is TRUE

  Syntax

  SELECT ColumnNames, aggregate_function (expression)
  FROM tables [WHERE conditions]
  GROUP BY column1 HAVING condition;

- Example:

  SELECT region, COUNT(customer_id) AS customer_count
  FROM customer GROUP BY region
  HAVING COUNT(customer_id) > 200;

➢ CASE statement:

- The CASE expression is a conditional expression, similar to if/else statements

  Syntax

  CASE WHEN condition THEN result
  [WHEN ...]
  [ELSE result]
  END ;

  CASE expression
  WHEN value THEN result
  [WHEN ...]
  [ELSE result]
  END;

- Example:

  SELECT *,
  CASE WHEN age<30 THEN 'Young'
  WHEN age>60 THEN 'Senior Citizen'
  ELSE 'Middle aged'
  END AS Age_category

  FROM customer;

➢ JOIN:

- JOINS are used to retrieve data from multiple tables. It is performed whenever two or more tables are joined in a SQL statement.

  TYPES

  - INNER JOIN (or sometimes called simple join)
  - LEFT OUTER JOIN (or sometimes called LEFT JOIN)
  - RIGHT OUTER JOIN (or sometimes called RIGHT JOIN)
  - FULL OUTER JOIN (or sometimes called FULL JOIN)
  - CROSS JOIN (or sometimes called CARTESIAN JOIN)

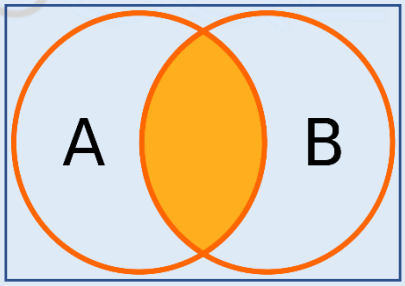
- INNER JOIN compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the joinpredicate. When satisfied, column values for each matched pair of rows of A and B are combined into a result row.

  Syntax

```
SELECT columns
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

- Example

```
SELECT
        a.order_line ,
        a.product_id,
        a.customer_id,
        a.sales,
        b.customer_name,
        b.age
FROM sales_2015 AS a
INNER JOIN customer_20_60 AS b
ON a.customer_id = b.customer_id
ORDER BY customer_id;
```



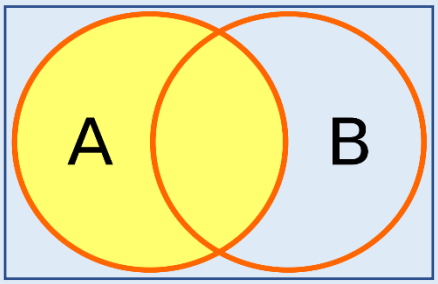
- The LEFT JOIN returns all rows from the left table, even if there are no matches in the right table.

  Syntax

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

- Example

```
SELECT
    a.order_line ,
    a.product_id,
    a.customer_id,
    a.sales,
    b.customer_name,
    b.age
FROM sales_2015 AS a
LEFT JOIN customer_20_60 AS b
ON a.customer_id = b.customer_id
ORDER BY customer_id;
```



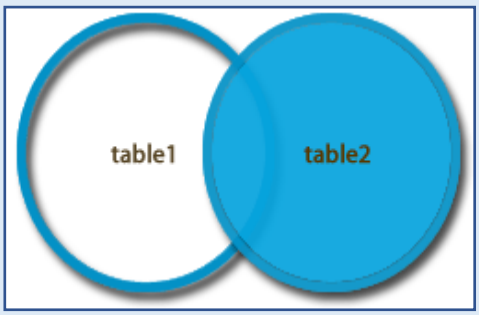
- The RIGHT JOIN returns all rows from the right table, even if there are no matches in the left table.

Syntax

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

- Example

```
SELECT
    a.order_line ,
    a.product_id,
    a.customer_id,
    a.sales,
    b.customer_name,
    b.age
    FROM sales_2015 AS a
    RIGHT JOIN customer_20_60 AS b
    ON a.customer_id = b.customer_id
    ORDER BY customer_id;
```



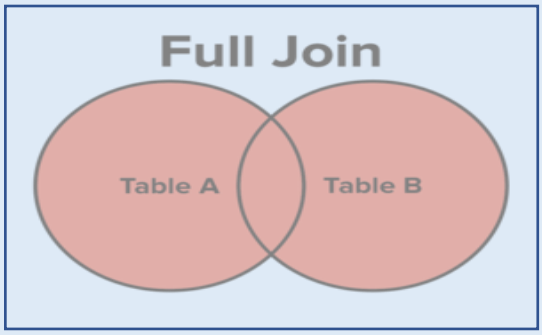
- The FULL JOIN combines the results of both left and right outer joins

Syntax

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

- Example

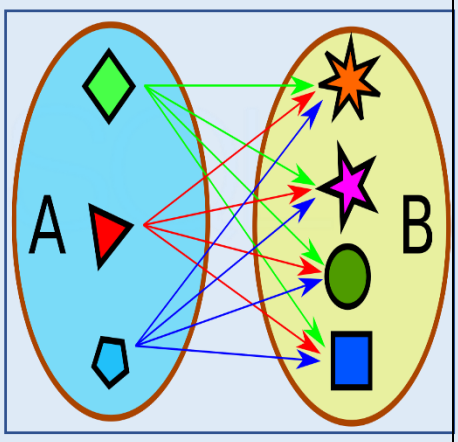
```
SELECT
    a.order_line ,
    a.product_id,
    a.customer_id,
    a.sales,
    b.customer_name,
    b.age,
    b.customer_id
    FROM sales_2015 AS a
    FULL JOIN customer_20_60 AS b
    ON a.customer_id = b.customer_id
    ORDER BY a.customer_id , b.customer_id;
```



- The Cross Join creates a cartesian product between two sets of data.

Syntax

```
SELECT table1.column1, table2.column2...
FROM table1, table2 [, table3 ];
```



- Example

```
SELECT a.YYYY, b.MM
FROM year_values AS a, month_values AS b
ORDER BY a.YYYY, b.MM;
```

➢ INTERSECT:

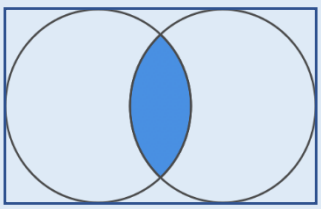
- Intersect operator is used to find the common rows from the result of two select queries

Syntax

```
SELECT column A, column B… FROM Table X
INTERSECT
SELECT column A, column B… FROM Table Y
```

- Example

```
SELECT customer_id  FROM sales_2015
INTERSECT
SELECT customer_id FROM customer_20_60
```



➢ Except

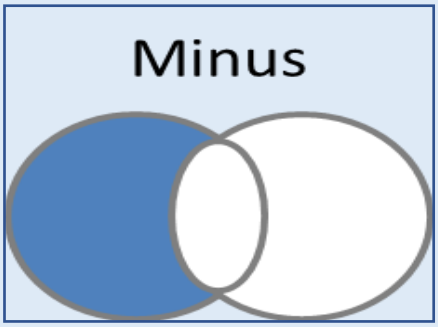
- EXCEPT operator is used to return all rows in the first SELECT statement that are not returned by the second SELECT statement.

Syntax

```
SELECT expression1, expression2, ...
FROM tables [WHERE conditions]
EXCEPT SELECT expression1, expression2, ...
FROM tables [WHERE conditions];
```

- Example

```
SELECT customer_id
FROM sales_2015
EXCEPT SELECT customer_id
FROM customer_20_60
ORDER BY customer_id;
```



➢ UNION

- UNION operator is used to combine the result sets of 2 or more SELECT statements. It removes duplicate rows between the various SELECT statements.

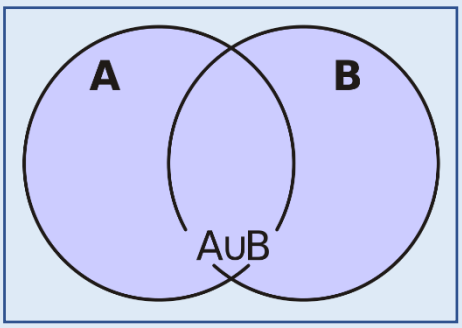
  Each SELECT statement within the UNION operator must have the same number of fields in the result sets with similar data types

Syntax

```
SELECT expression1, expression2, ... expression_n
FROM tables [WHERE conditions]
UNION SELECT expression1, expression2, ... expression_n
FROM tables [WHERE conditions];
```

- Example

```
SELECT customer_id
FROM sales_2015
UNION SELECT customer_id
FROM customer_20_60
ORDER BY customer_id;
```



- Note : INTERESECT ALL, UNIOUN ALL returns the duplicate value in output.

➢ Subquery

- Subquery is a query within a query. These subqueries can reside in the WHERE clause, the FROM clause, or the SELECT clause.

  Syntax

  - SYNTAX where subquery is in WHERE

```
SELECT "column_name1"
FROM "table_name1"
WHERE "column_name2" [Comparison Operator]
(SELECT "column_name3"
FROM "table_name2"
WHERE "condition");
```

  - Example

    Subquery in WHERE

```
SELECT * FROM sales
WHERE customer_ID IN
(SELECT DISTINCT customer_id
FROM customer
WHERE age >60);
```

  - Subquery in FROM

```
SELECT
        a.product_id ,
        a.product_name ,
        a.category,
        b.quantity
FROM product AS a
LEFT JOIN (SELECT product_id,
                SUM(quantity) AS quantity
                FROM sales
                GROUP BY product_id) AS b
ON a.product_id = b.product_id
ORDER BY b.quantity desc ;
```

- Subquery in SELECT

```
SELECT customer_id, order_line,
        (SELECT customer_name
                FROM customer
                WHERE sales.customer_id = customer.customer_id)
FROM sales
ORDER BY customer_id;
```

- There are a few rules that subqueries must follow –
  • Subqueries must be enclosed within parentheses.
  • A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
  • An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
  • Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.
  • The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.
  • A subquery cannot be immediately enclosed in a set function.
  • The BETWEEN operator cannot be used with a subquery. However, the BETWEEN operator can be used within the subquery.

➢ VIEW

- VIEW is not a physical table, it is a virtual table created by a query joining one or more tables.

Syntax

```
CREATE [OR REPLACE] VIEW view_name
AS SELECT columns
FROM tables [WHERE conditions];
```

- Example

```
CREATE VIEW logistics AS
SELECT a.order_line, a.order_id,
        b.customer_name, b.city,
        b. state, b. country
FROM sales AS a
LEFT JOIN customer as b
ON a.customer_id = b.customer_id
ORDER BY a.order_line;
```

CREATE OR REPLACE VIEW can be used instead of just CREATE VIEW

- DROP VIEW logistics;
- UPDATE logistics
  SET Country = US
  WHERE Country = 'United States';

- Notes
  A view is a virtual table. A view consists of rows and columns just like a table. The difference between a view and a table is that views are definitions built on top of other tables (or views), and do not hold data themselves. If data is changing in the underlying table, the same change is reflected in the view. A view can be built on top of a single table or multiple tables. It can also be built on top of another view. In the SQL Create View page, we will see how a view can be built. Views offer the following advantages:
  1. Ease of use: A view hides the complexity of the database tables from end users. Essentially we can think of views as a layer of abstraction on top of the database tables.
  2. Space savings: Views takes very little space to store, since they do not store actual data.
  3. Additional data security: Views can include only certain columns in the table so that only the non-sensitive columns are included and exposed to the end user. In addition, some databases allow views to have different security settings, thus hiding sensitive data from prying eyes.

- VIEW can be updated under certain conditions which are given below –
  • The SELECT clause may not contain the keyword DISTINCT.
  • The SELECT clause may not contain summary functions.
  • The SELECT clause may not contain set functions.
  • The SELECT clause may not contain set operators.
  • The SELECT clause may not contain an ORDER BY clause.
  • The FROM clause may not contain multiple tables.
  • The WHERE clause may not contain subqueries.
  • The query may not contain GROUP BY or HAVING.
  • Calculated columns may not be updated.
  • All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

➢ Index

- An index is a performance-tuning method of allowing faster retrieval of records. An index creates an entry for each value that appears in the indexed columns.

Syntax

```
CREATE [UNIQUE] INDEX index_name
ON table_name (index_col1 [ASC | DESC],
               index_col2 [ASC | DESC],
               ... index_col_n [ASC | DESC]);
```

A simple index is an index on a single column, while a composite index is an index on two or more columns

- Example

```
CREATE INDEX mon_idx
ON month_values(MM);
```

- DROP or RENAME

  Syntax

```
DROP INDEX [IF EXISTS] index_name
[ CASCADE | RESTRICT ];

ALTER INDEX [IF EXISTS] index_name,
RENAME TO new_index_name;
```

- Example

```
DROP INDEX mon_idx;
```

- Good practices
  1. Build index on columns of integer type
  2. Keep index as narrow as possible
  3. Column order is important
  4. Make sure the column you are building an index for is declared NOT NULL
  5. Build an index only when necessary

- The following guidelines indicate when the use of an index should be reconsidered.
  • Indexes should not be used on small tables.
  • Tables that have frequent, large batch updates or insert operations.
  • Indexes should not be used on columns that contain a high number of NULL values.
  • Columns that are frequently manipulated should not be indexed

➢ STRING FUNCTIONS

- LENGTH

  LENGTH function returns the length of the specified string, expressed as the number of characters.

  Syntax

  length(string)

```
SELECT Customer_name,
Length (Customer_name) as characters
FROM customer
WHERE age >30 ;
```

- UPPER/ LOWER

  UPPER/ LOWER function converts all characters in the specified string to uppercase/ lowercase.

  Syntax

  upper(string)
  lower(string)

```
SELECT upper('Start-Tech Academy');
SELECT lower('Start-Tech Academy');
```

- REPLACE

  REPLACE function replaces all occurrences of a specified string

  Syntax

  replace(string, from_substring, to_substring)

  Replace function is case sensitive

```
SELECT Customer_name, country,
Replace (country,'United States','US') AS country new
FROM customer;
```

- TRIM function

  TRIM function removes all specified characters either from the beginning or the end of a string

  RTRIM function removes all specified characters from the right-hand side of a string

  LTRIM function removes all specified characters from the left-hand side of a string

Syntax

```
trim( [ leading | trailing | both ] [ trim_character ] from string ) ;

rtrim( string, trim_character );

ltrim( string, trim_character );

SELECT
    trim(leading ' ' from ' Start-Tech Academy ');

SELECT
    trim(trailing ' ' from ' Start-Tech Academy ');

SELECT
    trim(both ' ' from ' Start-Tech Academy ');

SELECT
    trim(' Start-Tech Academy ');

SELECT
    rtrim(' Start-Tech Academy ', ' ');
SELECT ltrim(' Start-Tech Academy ', ' ');
```

- CONCAT
  || operator allows you to concatenate 2 or more strings together

Syntax

```
string1 || string2 || string_n

SELECT
    Customer_name, city|| ' , '||state|| ' , '||country AS address
    FROM customer;
```

- SUBSTRING
  SUBSTRING function allows you to extract a substring from a string

Syntax

```
substring( string [from start_position] [for length] )

SELECT Customer_id, Customer_name,
    SUBSTRING (Customer_id FOR 2) AS cust_group
    FROM customer
    WHERE SUBSTRING(Customer_id FOR 2) = 'AB';
```

```
SELECT
    Customer_id, Customer_name,
    SUBSTRING (Customer_id FROM 4 FOR 5) AS cust_number
    FROM customer
    WHERE SUBSTRING(Customer_id FOR 2) = 'AB';
```

- STRING AGGREGATOR
  STRING_AGG concatenates input values into a string, separated by delimiter

```
SELECT
    order_id ,
    STRING_AGG (product_id,', ')
FROM sales GROUP BY order_id;
```

➢ Mathematical function

- CEIL or FLOOR
  CEIL function returns the smallest integer value that is greater than or equal to a number
  FLOOR function returns the largest integer value that is equal to or less than a number.

Syntax

```
CEIL (number)
FLOOR (number)
```

Example

```
SELECT order_line,
sales, CEIL (sales),
FLOOR (sales)
FROM sales
WHERE discount>0;
```

- Random
  RANDOM function can be used to return a random number between 0 and 1

Syntax

```
RANDOM( )
```

The random function will return a value between 0 (inclusive) and 1 (exclusive), so value >= 0 and value < 1.

Example
Random decimal between a range (a included and b excluded)

```
SELECT RANDOM()*(b-a)+a
```

Random Integer between a range (both boundaries included)

```
SELECT FLOOR(RANDOM()*(b-a+1))+a;
```

- SET SEED

  If we set the seed by calling the setseed function, then the random function will return a repeatable sequence of random numbers that is derived from the seed.

  Syntax SETSEED (seed)

  Seed can have a value between 1.0 and -1.0, inclusive.

  Example

```
SELECT SETSEED(0.5);
SELECT RANDOM();
SELECT RANDOM();
```

- ROUND

  ROUND function returns a number rounded to a certain number of decimal places

  Syntax

  ROUND (number)

  Example

```
SELECT order_line, sales,
ROUND (sales)
FROM sales;
```

- Power

  POWER function returns m raised to the nth power

  Syntax

  POWER (m, n)

  This will be equivalent to m raised to the power n.

  Example

```
SELECT POWER(6, 2);
SELECT age, power(age,2)
FROM customer
ORDER BY age;
```

➢ DATE and TIME function

- Current date and time
  CURRENT_DATE function returns the current date.
  CURRENT_TIME function returns the current time with the time zone.
  CURRENT_TIMESTAMP function returns the current date and time with the time zone.

Syntax

```
CURRENT_DATE
CURRENT_TIME ( [ precision ] )
CURRENT_TIMESTAMP ( [ precision ] )
```

  - The CURRENT_DATE function will return the current date as a 'YYYY-MM-DD' format.
  - CURRENT_TIME function will return the current time of day as a 'HH:MM:SS.GMT+TZ' format.
  - The CURRENT_TIMESTAMP function will return the current date as a 'YYYYMM-DD HH:MM:SS.GMT+TZ' format.

Example

```
SELECT CURRENT_DATE;
SELECT CURRENT_TIME;
SELECT CURRENT_TIME(1);
SELECT CURRENT_TIMESTAMP;
```

- Age
  AGE function returns the number of years, months, and days between two dates.

Syntax

```
age( [date1,] date2 )
```

If date1 is NOT provided, current date will be used

Example

```
SELECT age('2014-04-25', '2014-01-01');

SELECT order_line, order_date, ship_date,
        age(ship_date, order_date) as time_taken
    FROM sales
    ORDER BY time_taken DESC;
```

- Extract

  EXTRACT function extracts parts from a date

  Syntax

  EXTRACT ('unit' from 'date')

  Example

  SELECT EXTRACT(day from '2014-04-25');
  SELECT EXTRACT(day from '2014-04-25 08:44:21');
  SELECT EXTRACT(minute from '08:44:21');

  SELECT order_line,
  EXTRACT(EPOCH FROM (ship_date - order_date))
  FROM sales;

- Units

Unit	Explanation
day	Day of the month (1 to 31)
decade	Year divided by 10
doy	Day of the year (1=first day of year, 365/366=last day of the year, depending if it is a leap year)
epoch	Number of seconds since '1970-01-01 00:00:00 UTC', if date value. Number of seconds in an interval, if interval value
hour	Hour (0 to 23)
minute	Minute (0 to 59)
month	Number for the month (1 to 12), if date value. Number of months (0 to 11), if interval value
second	Seconds (and fractional seconds)
year	Year as 4-digits

➢ Pattern Matching

- 1. LIKE statements 2. SIMILAR TO statements 3. ~ (Regular Expressions)
- ~ OPERATOR or
- REG-EX Wildcards

Wildcard	Explanation
|	Denotes alternation (either of two alternatives).
*	Denotes repetition of the previous item zero or more times
+	Denotes repetition of the previous item one or more times.
?	Denotes repetition of the previous item zero or one time.
{m}	denotes repetition of the previous item exactly m times.

{m,}	denotes repetition of the previous item m or more times.
{m,n}	denotes repetition of the previous item at least m and not more than n times
^,$	^ denotes start of the string, $ denotes end of the string
[chars]	a bracket expression, matching any one of the chars
~*	~means case sensitive and ~* means case insensitive

- Example

```
SELECT * FROM customer
WHERE customer_name ~* '^a+[a-z\s]+$' ;

SELECT * FROM customer
WHERE customer_name ~* '^(a|b|c|d)+[a-z\s]+$' ;

SELECT * FROM customer
WHERE customer_name ~* '^(a|b|c|d)[a-z]{3}\s[a-z]{4}$‘ ;

SELECT * FROM users
WHERE name ~* '[a-z0-9\.\-\_]+@[a-z0-9\-]+\.[a-z]{2,5}';
```

➢ Conversion To string

- Conversion to text
  TO_CHAR function converts a number or date to a string
  Syntax
  TO_CHAR (value, format_mask)
- Format Mask

Parameter	Explanation
9	Value (with no leading zeros)
0	Value (with leading zeros)
.	Decimal
,	Group separator
PR	Negative value in angle brackets
S	Sign
L	Currency symbol
MI	Minus sign (for negative numbers)
PL	Plus sign (for positive numbers)
SG	Plus/minus sign (for positive and negative numbers)
EEEE	Scientific notation

- Format Mask

Parameter	Explanation
YYYY	4-digit year
MM	Month (01-12; JAN = 01).
Mon	Abbreviated name of month capitalized
Month	Name of month capitalized, padded with blanks to length of 9 characters
DAY	Name of day in all uppercase, padded with blanks to length of 9 characters
Day	Name of day capitalized, padded with blanks to length of 9 characters
DDD	Day of year (1-366)
DD	Day of month (01-31)
HH	Hour of day (01-12)
HH12	Hour of day (01-12)
HH24	Hour of day (00-23)
MI	Minute (00-59)
SS	Second (00-59)
am, AM, pm, or PM	Meridian indicator

- Example

```
SELECT sales, TO_CHAR(sales, '9999.99')
FROM sales;

SELECT sales, TO_CHAR(sales, 'L9,999.99')
FROM sales;

SELECT order_date, TO_CHAR(order_date, 'MMDDYY')
FROM sales;

SELECT order_date, TO_CHAR(order_date, 'Month DD, YYYY')
FROM sales;
```

- CONVERSION TO DATE

  TO_DATE function converts a string to a date.

  Syntax

  ```
  TO_DATE( string1, format_mask )
  ```

- Example

```
SELECT TO_DATE('2014/04/25', 'YYYY/MM/DD');
SELECT TO_DATE('033114', 'MMDDYY');
```

- CONVERSION TO NUMBER
  TO_NUMBER function converts a string to a number

  Syntax

```
TO_NUMBER( string1, format_mask )
```

- Example

```
SELECT TO_NUMBER ('1210.73', '9999.99');
SELECT TO_NUMBER ('$1,210.73', 'L9,999.99');
```

➢ USER ACCESS CONTROL

- CREATE USER
  CREATE USER statement creates a database account that allows you to log into the database

  Syntax

```
CREATE USER user_name
[WITH PASSWORD 'password_value'
| VALID UNTIL 'expiration' ];
```

- Example

```
CREATE USER starttech
WITH PASSWORD 'academy';

CREATE USER starttech
WITH PASSWORD 'academy '
VALID UNTIL 'Jan 1, 2020';

CREATE USER starttech
WITH PASSWORD 'academy '
VALID UNTIL 'infinity';
```

- GRANT & REVOKE
  Privileges to tables can controlled using GRANT & REVOKE. These permissions can be any combination of SELECT, INSERT, UPDATE, DELETE, INDEX, CREATE, ALTER, DROP, GRANT OPTION or ALL.

Syntax

```
GRANT privileges ON object TO user;
REVOKE privileges ON object FROM user;
```

- PRIVILEGE

Privilege	Description
SELECT	Ability to perform SELECT statements on the table.
INSERT	Ability to perform INSERT statements on the table.
UPDATE	Ability to perform UPDATE statements on the table.
DELETE	Ability to perform DELETE statements on the table.
TRUNCATE	Ability to perform TRUNCATE statements on the table.
REFERENCES	Ability to create foreign keys (requires privileges on both parent and child tables).
TRIGGER	Ability to create triggers on the table.
CREATE	Ability to perform CREATE TABLE statements.
ALL	Grants all permissions.

- Example

```
GRANT SELECT, INSERT, UPDATE, DELETE ON products
TO starttech;

GRANT ALL ON products TO starttech;

GRANT SELECT ON products TO PUBLIC;
REVOKE ALL ON products FROM starttech;
```

- DROP user

  DROP USER statement is used to remove a user from the database.

  Syntax

  ```
  DROP USER user_name;
  ```

  If the user that you wish to delete owns a database, be sure to drop the database first and then drop the user.

- Example

  ```
  DROP USER techonthenet;
  ```

- RENAME user

  ALTER USER statement is used to rename a user in the database

Syntax

```
ALTER USER user_name
RENAME TO new_name;
```

- Example

```
ALTER USER starttech
RENAME TO ST;
```

- FIND ALL User

  Run a query against pg_user table to retrieve information about Users

  Syntax

```
SELECT usename FROM pg_user;
```

- FIND ALL Logged in user

  Run a query against pg_stat_activity table to retrieve information about Logged-in Users

  Syntax

```
SELECT DISTINCT usename
FROM pg_stat_activity;
```

➢ TABLESPACES

- Tablespaces allow database administrators to define locations in the file system where the files representing database objects can be stored

  Syntax

```
CREATE TABLESPACE LOCATION ;
```

• Creation of the tablespace can only be done by database superuser

• Ordinary database users can be allowed to use it by granting them the CREATE privilege on the new tablespace

- Example

```
CREATE TABLESPACE newspace
LOCATION '/mnt/sda1/postgresql/data';

CREATE TABLE first_table (test_column int)
TABLESPACE newspace;

SET default_tablespace = newspace;
```

CREATE TABLE second_table(test_column int);

SELECT newspace FROM pg_tablespace;

- USES
  • If the partition or volume on which the cluster was initialized runs out of space and cannot be extended, a tablespace can be created on a different partition and used until the system can be reconfigured.
  • Tablespaces allow an administrator to use knowledge of the usage pattern of database objects to optimize performance. For example, an index which is very heavily used can be placed on a very fast, highly available disk, such as an expensive solid state device. At the same time a table storing archived data which is rarely used or not performance critical could be stored on a less expensive, slower disk system

➢ Primary Key

➢Primary key consists of one or more columns

➢Used to uniquely identify each row in the table

➢No value in the columns can be blank or NULL

➢ ACID

- ACID (an acronym for Atomicity, Consistency Isolation, Durability) is a concept that Database Professionals generally look for when evaluating databases and application architectures. For a reliable database all these four attributes should be achieved.
  ➢ ATOMICITY Atomicity is an all-or-none proposition
  ➢ CONSISTENCY Consistency ensures that a transaction can only bring the database from one valid state to another
  ➢ ISOLATION Isolation keeps transactions separated from each other until they're finished.
  ➢ DURABILITY Durability guarantees that the database will keep track of pending changes in such a way that the server can recover from an abnormal termination

➢ TRUNCATE

- The TRUNCATE TABLE statement is used to remove all records from a table or set of tables in PostgreSQL. It performs the same function as a DELETE statement without a WHERE clause.

  Syntax

  ```
  TRUNCATE [ONLY] table_name
  [ CASCADE | RESTRICT];
  ```

- Example

  ```
  TRUNCATE TABLE Customer_20_60;

  Same as
  DELETE FROM Customer_20_60;
  ```

➢ Best Practices

- Displays the execution plan for a query statement without running the query.

  ```
  EXPLAIN [ VERBOSE ] query;
  ```

  VERBOSE

  Displays the full query plan instead of just a summary.

  query

  Query statement to explain.

- SOFT DELETE vs HARD DELETE

  SOFT DELETE

  Soft deletion means you don't actually delete the record instead you are marking the record as deleted

  HARD DELETE

  Hard deletion means data is physically deleted from the database table.

- UPDATE vs CASE

  UPDATE

  Update customer set customer_name = (trim(upper(customer_name)) where (trim(upper(customer_name)) <> customer_name

  Every updated row is actually a soft delete and an insert. So updating every row will increase the storage size of the table

  CASE STATEMENT

  Instead you can use the case statements while creating such tables

- VACCUM

  SYNTAX

  VACUUM [table] ;

  USE

  ➢ Reclaims disk space occupied by rows that were marked for deletion by previous UPDATE and DELETE operations.

  ➢ Compacts the table to free up the consumed space

  ➢ Use it on tables which you are updating and deleting on a regular basis

- TRUNCATE VS DELETE

  ➢ The TRUNCATE statement is typically far more efficient than using the DELETE statement with no WHERE clause to empty the table

  ➢ TRUNCATE requires fewer resources and less logging overhead

  ➢ Instead of creating table each time try to use truncate as it will keep the table structure and properties intact

  ➢ Truncate frees up space and impossible to rollback

- STRING FUNCTIONS

  Pattern Matching

  ➢ Whenever possible use LIKE statements in place of REGEX expressions

  ➢ Do not use 'Similar To' statements, instead use Like and Regex

  ➢ Avoid unnecessary string operations such as replace, upper, lower etc String Operations

  ➢ Use trim instead of replace whenever possible

  ➢ Avoid unnecessary String columns. For eg. Use date formats instead of string for dates

- JOINS

  Syntax

```
SELECT a.order_line , a.product_id, b.customer_name, b.age
FROM sales_2015 AS a LEFT JOIN customer_20_60 AS b
ON a.customer_id = b.customer_id
ORDER BY customer_id;
```

Best Practices

➢ Use subqueries to select only the required fields from the tables

➢ Avoid one to many joins by mentioning Group by clause on the matching fields

- SCHEMAS

  A schema is a collection of database objects associated with one particular database. You may have one or multiple schemas in a database.

  1. To allow many users to use one database without interfering with each other.
  2. To organize database objects into logical groups to make them more manageable.
  3. Third-party applications can be put into separate schemas so they do not collide with the names of other objects.

  SCHEMAS (Syntax)

  CREATE SCHEMA testschema;

➢ SQL

- What is SQL

  ➢ Computer Language used for
  - Storing
  - Manipulating
  - Retrieving Data

  ➢ Invented by IBM

  ➢ SQL stands for Structured Query Language

- Why SQL

  Large Amount of Data
  Controlled access
  Data Manipulation
  Business Insights

- Who uses SQL

  Software developers
  Database Managers
  Business Managers

➢ DBMS Database Management Systems

- ➢It allows creation of new DB and their data structures
  ➢Allows modification of data
  ➢Allows retrieval of Data
  ➢Allows Storage over long period of time
  ➢Enables recovery in times of failure
  ➢Control access to users

➢ SQL QUERIES

1. DDL – Data Definition Language
   CREATE, ALTER, DROP

2. DML – Data Manipulation Language
   INSERT, UPDATE, DELETE

3. DQL – Data Query Language
   SELECT, ORDER BY, GROUP BY

4. DCL – Data Control Language
   GRANT, REVOKE

5. TCC – Transactional Control Commands
   COMMIT, ROLLBACK

➢ What is PostgreSQL

PostgreSQL is an advanced object-relational database management system that supports an extended subset of the SQL standard, including transactions, foreign keys, subqueries, triggers, user-defined types and functions.

➢ Why PostgreSQL

➢ Completely Open source

➢ Complete ACID Compliance

➢ Comprehensive documentation and active discussion forums

➢ PostgreSQL performance is utilized best in systems requiring execution of complex queries

➢ PostgreSQL is best suited for Data Warehousing and data analysis applications that require fast read/write speeds

➢ Supported by all major cloud service providers, including Amazon, Google, & Microsoft