SQL

MOST IMPORTANT CONCEPTS

PLACEMENT PREPARATION

[EXCLUSIVE NOTES]

[SAVE AND SHARE]

Curated By- HIMANSHU KUMAR(LINKEDIN)

TOPICS COVERED-

PART-2 :-

- Constraints
- Comments
- GROUP BY
- Views
- Functions (Aggregate and Scalar Functions)
- Query Processing
- WHERE Clause

HIMANSHU KUMAR(LINKEDIN)

https://www.linkedin.com/in/himanshukumarmahuri

Constraints-

Constraints are the rules that we can apply on the type of data in a table. That is, we can specify the limit on the type of data that can be stored in a particular column in a table using constraints.

The available constraints in SQL are:

- **NOT NULL**: This constraint tells that we cannot store a null value in a column. That is, if a column is specified as NOT NULL then we will not be able to store null in this particular column any more.
- **UNIQUE**: This constraint when specified with a column, tells that all the values in the column must be unique. That is, the values in any row of a column must not be repeated.
- **PRIMARY KEY**: A primary key is a field which can uniquely identify each row in a table. And this constraint is used to specify a field in a table as primary key.
- **FOREIGN KEY**: A Foreign key is a field which can uniquely identify each row in a another table. And this constraint is used to specify a field as Foreign key.
- **CHECK**: This constraint helps to validate the values of a column to meet a particular condition. That is, it helps to ensure that the value stored in a column meets a specific condition.
- **DEFAULT**: This constraint specifies a default value for the column when no value is specified by the user.

How to specify constraints?

We can specify constraints at the time of creating the table using CREATE TABLE statement. We can also specify the constraints after creating a table using ALTER TABLE statement.

Syntax:

Below is the syntax to create constraints using CREATE TABLE statement at the time of creating the table.

```
CREATE TABLE sample_table
(
column1 data_type(size) constraint_name,

column2 data_type(size) constraint_name,

column3 data_type(size) constraint_name,
....
);
sample_table: Name of the table to be created.
data_type: Type of data that can be stored in the field.
constraint_name: Name of the constraint. for example- NOT NULL, UNIQUE, PRIMARY KEY etc.
```

Let us see each of the constraint in detail.

1. NOT NULL -

If we specify a field in a table to be NOT NULL. Then the field will never accept null value. That is, you will be not allowed to insert a new row in the table without specifying any value to

this field.
For example, the below query creates a table Student with the fields ID and NAME as NOT NULL. That is, we are bound to specify values for these two fields every time we wish to insert a new row.

```
CREATE TABLE Student
(
ID int(6) NOT NULL,
NAME varchar(10) NOT NULL,
ADDRESS varchar(20)
);
```

2. UNIQUE -

This constraint helps to uniquely identify each row in the table. i.e. for a particular column, all the rows should have unique values. We can have more than one UNIQUE columns in a table.
For example, the below query creates a table Student where the field ID is specified as UNIQUE. i.e, no two students can have the same ID.

```
CREATE TABLE Student
(
ID int(6) NOT NULL UNIQUE,
NAME varchar(10),
ADDRESS varchar(20)
);
```

3. PRIMARY KEY -

Primary Key is a field which uniquely identifies each row in the table. If a field in a table as primary key, then the field will not be able to contain NULL values as well as all the rows should have unique values for this field. So, in other words we can say that this is combination of NOT NULL and UNIQUE constraints.

A table can have only one field as primary key. Below query will create a table named Student and specifies the field ID as primary key.

```
CREATE TABLE Student
(
ID int(6) NOT NULL UNIQUE,
NAME varchar(10),
ADDRESS varchar(20),
PRIMARY KEY(ID)
);
```

4. FOREIGN KEY -

Foreign Key is a field in a table which uniquely identifies each row of a another table. That is, this field points to primary key of another table. This usually creates a kind of link between the tables.

Consider the two tables as shown below:

Orders

O_ID	ORDER_NO	C_ID

1	2253	3
2	3325	3
3	4521	2
4	8532	1

Customers

C_ID	NAME	ADDRESS
1	RAMESH	DELHI
2	SURESH	NOIDA
3	DHARMESH	GURGAON

As we can see clearly that the field C_ID in Orders table is the primary key in Customers table, i.e. it uniquely identifies each row in the Customers table. Therefore, it is a Foreign Key in Orders table.

Syntax:

```
CREATE TABLE Orders
(
O_ID int NOT NULL,
ORDER_NO int NOT NULL,
```

```
C_ID int,
PRIMARY KEY (O_ID),
FOREIGN KEY (C_ID) REFERENCES Customers(C_ID)
)
```

(i) CHECK -

Using the CHECK constraint we can specify a condition for a field, which should be satisfied at the time of entering values for this field.

For example, the below query creates a table Student and specifies the condition for the field AGE as (AGE >= 18). That is, the user will not be allowed to enter any record in the table with AGE < 18.

```
CREATE TABLE Student
(
ID int(6) NOT NULL,
NAME varchar(10) NOT NULL,
AGE int NOT NULL CHECK (AGE >= 18)
);
```

(ii) DEFAULT -

This constraint is used to provide a default value for the fields. That is, if at the time of entering new records in the table if the user does not specify any value for these fields then the default value will be assigned to them.

For example, the below query will create a table named Student and specify the default value for the field AGE as 18.

```
CREATE TABLE Student
(
```

```
ID int(6) NOT NULL,
NAME varchar(10) NOT NULL
AGE int DEFAULT 18
);
```

Comments-

As is any programming languages comments matter a lot in SQL also. In this set we will learn about writing comments in any SQL snippet.

Comments can be written in the following three formats:

1. Single line comments.
2. Multi line comments
3. In line comments

- **Single line comments:** Comments starting and ending in a single line are considered as single line comments. Line starting with '--' is a comment and will not be executed.
  Syntax:

```
-- single line comment
-- another comment
SELECT * FROM Customers;
```

- **Multi line comments:** Comments starting in one line and ending in different line are considered as multi line comments. Line starting with '/*' is considered as starting point of comment and are terminated when '*/' is encountered.
  Syntax:

```
/* multi line comment
another comment */
SELECT * FROM Customers;
```

- **In line comments:** In line comments are an extension of multi line comments, comments can be stated in between the statements and are enclosed in between '/*' and '*/'.
  Syntax:

```
SELECT * FROM /* Customers; */
```

More examples:

```
Multi line comment ->
/* SELECT * FROM Students;
SELECT * FROM STUDENT_DETAILS;
SELECT * FROM Orders; */
SELECT * FROM Articles;
In line comment ->
SELECT * FROM Students;
SELECT * FROM /* STUDENT_DETAILS;
SELECT * FROM Orders;
SELECT * FROM */ Articles;
```

GROUP BY-

The GROUP BY Statement in SQL is used to arrange identical data into groups with the help of some functions. i.e if a particular column has same values in different rows then it will arrange these rows in a group.

Important Points:

- GROUP BY clause is used with the SELECT statement.
- In the query, GROUP BY clause is placed after the WHERE clause.
- In the query, GROUP BY clause is placed before ORDER BY clause if used any.

Syntax:

```
SELECT column1, function_name(column2)
FROM table_name
WHERE condition
GROUP BY column1, column2
ORDER BY column1, column2;
function_name: Name of the function used for example, SUM( ) , AVG().
table_name: Name of the table.
condition: Condition used.
```

Sample Table:
Employee

SI NO	NAME	SALARY	AGE
1	Harsh	2000	19
2	Dhanraj	3000	20
3	Ashish	1500	19
4	Harsh	3500	19
5	Ashish	1500	19

Student

SUBJECT	YEAR	NAME
English	1	Harsh
English	1	Pratik
English	1	Ramesh
English	2	Ashish
English	2	Suresh
Mathematics	1	Deepak
Mathematics	1	Sayan

Example:

- **Group By single column**: Group By single column means, to place all the rows with same value of only that particular column in one group. Consider the query as shown below:

```
SELECT NAME, SUM(SALARY) FROM Employee
GROUP BY NAME;
```

The above query will produce the below output:

NAME	SALARY
Ashish	3000
Dhanraj	3000
Harsh	5500

As you can see in the above output, the rows with duplicate NAMEs are grouped under same NAME and their corresponding SALARY is the sum of the SALARY of duplicate rows. The SUM() function of SQL is used here to calculate the sum.

- **Group By multiple columns**: Group by multiple column is say for example, **GROUP BY column1, column2**. This means to place all the rows with same values of both the columns **column1** and **column2** in one group. Consider the below query:

```
SELECT SUBJECT, YEAR, Count(*)
FROM Student
GROUP BY SUBJECT, YEAR;
```

Output:

SUBJECT	YEAR	Count
English	1	3
English	2	2
Mathematics	1	2

As you can see in the above output the students with both same SUBJECT and YEAR are placed in same group. And those whose only SUBJECT is same but not YEAR belongs to different groups. So here we have grouped the table according to two columns or more than one column.

HAVING Clause

We know that WHERE clause is used to place conditions on columns but what if we want to place conditions on groups?

This is where HAVING clause comes into use. We can use HAVING clause to place conditions to decide which group will be the part of final result-set. Also we can not use the aggregate functions like SUM(), COUNT() etc. with WHERE clause. So we have to use HAVING clause if we want to use any of these functions in the conditions.

Syntax:

```
SELECT column1, function_name(column2)
FROM table_name
WHERE condition
GROUP BY column1, column2
HAVING condition
ORDER BY column1, column2;
function_name: Name of the function used for example, SUM( ) , AVG().
table_name: Name of the table.
condition: Condition used.
```

Example:

```
SELECT NAME, SUM(SALARY) FROM Employee
GROUP BY NAME
HAVING SUM(SALARY)>3000;
```

Output:

NAME	SUM(SALARY)
HARSH	5500

As you can see in the above output only one group out of the three groups appears in the result-set as it is the only group where sum of SALARY is greater than 3000. So we have used HAVING clause here to place this condition as the condition is required to be placed on groups not columns.

Views-

Views in SQL are kind of virtual tables. A view also has rows and columns as they are in a real table in the database. We can create a view by selecting fields from one or more tables present in the database. A View can either have all the rows of a table or specific rows based on certain condition.

In this article we will learn about creating , deleting and updating Views.

Sample Tables:

StudentDetails

S_ID	NAME	ADDRESS
1	Harsh	Kolkata
2	Ashish	Durgapur
3	Pratik	Delhi
4	Dhanraj	Bihar
5	Ram	Rajasthan

StudentMarks

ID	NAME	MARKS	AGE
1	Harsh	90	19
2	Suresh	50	20
3	Pratik	80	19
4	Dhanraj	95	21
5	Ram	85	18

CREATING VIEWS

We can create View using **CREATE VIEW** statement. A View can be created from a single table or multiple tables.

Syntax:

```
CREATE VIEW view_name AS
SELECT column1, column2.....
```

```
FROM table_name
WHERE condition;
view_name: Name for the View
table_name: Name of the table
condition: Condition to select rows
```

Examples:

- **Creating View from a single table:**
  - In this example we will create a View named DetailsView from the table StudentDetails.
    Query:

```
CREATE VIEW DetailsView AS
SELECT NAME, ADDRESS
FROM StudentDetails
WHERE S_ID < 5;
```

To see the data in the View, we can query the view in the same manner as we query a table.

```
SELECT * FROM DetailsView;
```

Output:

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

In this example, we will create a view named StudentNames from the table StudentDetails.

Query:

```
CREATE VIEW StudentNames AS
SELECT S_ID, NAME
FROM StudentDetails
ORDER BY NAME;
```

If we now query the view as,

```
SELECT * FROM StudentNames;
```

Output:

S_ID	NAMES
2	Ashish
4	Dhanraj
1	Harsh
3	Pratik
5	Ram

- **Creating View from multiple tables**: In this example we will create a View named MarksView from two tables StudentDetails and StudentMarks. To create a View from multiple tables we can simply include multiple tables in the SELECT statement. Query:

```
CREATE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS, S
tudentMarks.MARK
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;
```

To display data of View MarksView:

```
SELECT * FROM MarksView;
```

Output:

NAME	ADDRESS	MARKS
Harsh	Kolkata	90
Pratik	Delhi	80
Dhanraj	Bihar	95
Ram	Rajasthan	85

DELETING VIEWS

We have learned about creating a View, but what if a created View is not needed any more? Obviously we will want to delete it. SQL allows us to delete an existing View. We can delete or drop a View using the DROP statement.

Syntax:

```
DROP VIEW view_name;
view_name: Name of the View which we want to delete.
```

For example, if we want to delete the View **MarksView**, we can do this as:

```
DROP VIEW MarksView;
```

UPDATING VIEWS

There are certain conditions needed to be satisfied to update a view. If any one of these conditions is **not** met, then we will not be allowed to update the view.

1. The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
2. The SELECT statement should not have the DISTINCT keyword.
3. The View should have all NOT NULL values.
4. The view should not be created using nested queries or complex queries.
5. The view should be created from a single table. If the view is created using multiple tables then we will not be allowed to update the view.

We can use the **CREATE OR REPLACE VIEW** statement to add or remove fields from a view.
Syntax:

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1,coulmn2,..
FROM table_name
WHERE condition;
```

For example, if we want to update the view **MarksView** and add the field AGE to this View from **StudentMarks** Table, we can do this as:

```
CREATE OR REPLACE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS, StudentMarks.AGE
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;
```

If we fetch all the data from MarksView now as:

```
SELECT * FROM MarksView;
```

Output:

NAME	ADDRESS	MARKS	AGE
Harsh	Kolkata	90	19
Pratik	Delhi	80	19
Dhanraj	Bihar	95	21
Ram	Rajasthan	85	18

Inserting a row in a view:
We can insert a row in a View in a same way as we do in a table. We can use the INSERT INTO statement of SQL to insert a row in a View.**Syntax**:

```
INSERT INTO view_name(column1, column2 , column3,..)
VALUES(value1, value2, value3..);
view_name: Name of the View
```

Example:
In the below example we will insert a new row in the View DetailsView which we have created above in the example of "creating views from a single table".

```
INSERT INTO DetailsView(NAME, ADDRESS)
VALUES("Suresh","Gurgaon");
```

If we fetch all the data from DetailsView now as,

```
SELECT * FROM DetailsView;
```

Output:

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar
Suresh	Gurgaon

Deleting a row from a View:

Deleting rows from a view is also as simple as deleting rows from a table. We can use the DELETE statement of SQL to delete rows from a view. Also deleting a row from a view first delete the row from the

actual table and the change is then reflected in the view.**Syntax**:

```
DELETE FROM view_name
WHERE condition;
```

- **view_name**:Name of view from where we want to delete rows
- **condition**: Condition to select rows

Example:
In this example we will delete the last row from the view DetailsView which we just added in the above example of inserting rows.

```
DELETE FROM DetailsView
WHERE NAME="Suresh";
```

If we fetch all the data from DetailsView now as,

```
SELECT * FROM DetailsView;
```

Output:

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

WITH CHECK OPTION

The WITH CHECK OPTION clause in SQL is a very useful clause for views. It is applicable to a updatable view. If the view is not updatable, then there is no meaning of including this clause in the CREATE VIEW statement.

- The WITH CHECK OPTION clause is used to prevent the insertion of rows in the view where the condition in the WHERE clause in CREATE VIEW statement is not satisfied.

- If we have used the WITH CHECK OPTION clause in the CREATE VIEW statement, and if the UPDATE or INSERT clause does not satisfy the conditions then they will return an error.

Example:
In the below example we are creating a View SampleView from StudentDetails Table with WITH CHECK OPTION clause.

```
CREATE VIEW SampleView AS
SELECT S_ID, NAME
FROM  StudentDetails
WHERE NAME IS NOT NULL
WITH CHECK OPTION;
```

In this View if we now try to insert a new row with null value in the NAME column then it will give an error because the view is created with the condition for NAME column as NOT NULL.
For example,though the View is updatable but then also the below query for this View is not valid:

```
INSERT INTO SampleView(S_ID)
VALUES(6);
```

NOTE: The default value of NAME column is *null*.

Uses of a View : A good database should contain views due to the given reasons:

1. **Restricting data access -** Views provide an additional level of table security by restricting access to a predetermined set of rows and columns of a table.
2. **Hiding data complexity -** A view can hide the complexity that exists in a multiple table join.
3. **Simplify commands for the user -** Views allows the user to select information from multiple tables without requiring the users to actually know how to perform a join.

4. **Store complex queries -** Views can be used to store complex queries.
5. **Rename Columns -** Views can also be used to rename the columns without affecting the base tables provided the number of columns in view must match the number of columns specified in select statement. Thus, renaming helps to hide the names of the columns of the base tables.
6. **Multiple view facility -** Different views can be created on the same table for different users.

Functions (Aggregate and Scalar Functions)-

For doing operations on data SQL has many built-in functions, they are categorized in two categories and further sub-categorized in different seven functions under each category. The categories are:

1. **Aggregate functions:**
   These functions are used to do operations from the values of the column and a single value is returned.
   0. AVG()
   1. COUNT()
   2. FIRST()
   3. LAST()
   4. MAX()
   5. MIN()
   6. SUM()
2. **Scalar functions:**
   These functions are based on user input, these too returns single value.
   0. UCASE()

1. LCASE()
2. MID()
3. LEN()
4. ROUND()
5. NOW()
6. FORMAT()

Students-Table

ID	NAME	MARKS	AGE
1	Harsh	90	19
2	Suresh	50	20
3	Pratik	80	19
4	Dhanraj	95	21
5	Ram	85	18

Aggregate Functions

AVG(): It returns the average value after calculating from values in a numeric column.

Syntax:

```
SELECT AVG(column_name) FROM table_name;
```

Queries:

Computing average marks of students.

```
SELECT AVG(MARKS) AS AvgMarks FROM Students;
```

Output:

AvgMarks
80

Computing average age of students.

```
SELECT AVG(AGE) AS AvgAge FROM Students;
```

Output:

AvgAge
19.4

COUNT(): It is used to count the number of rows returned in a SELECT statement. It can't be used in MS ACCESS.

Syntax:

```
SELECT COUNT(column_name) FROM table_name;
```

Queries:

- Computing total number of students.

```
SELECT COUNT(*) AS NumStudents FROM Students;
```

Output:

NumStudents
5

- Computing number of students with unique/distinct age.

```
SELECT COUNT(DISTINCT AGE) AS NumStudents FROM Stud
ents;
```

Output:

NumStudents
4

FIRST(): The FIRST() function returns the first value of the selected column.

Syntax:

```
SELECT FIRST(column_name) FROM table_name;
```

Queries:

- Fetching marks of first student from the Students table.

```
SELECT FIRST(MARKS) AS MarksFirst FROM Students;
```

Output:

MarksFirst
90

- Fetching age of first student from the Students table.

```
SELECT FIRST(AGE) AS AgeFirst FROM Students;
```

Output:

AgeFirst
19

LAST(): The LAST() function returns the last value of the selected column. It can be used only in MS ACCESS.

Syntax:

```
SELECT LAST(column_name) FROM table_name;
```

Queries:

Fetching marks of last student from the Students table.

```
SELECT LAST(MARKS) AS MarksLast FROM Students;
```

Output:

MarksLast
82

- Fetching age of last student from the Students table.

```
SELECT LAST(AGE) AS AgeLast FROM Students;
```

Output:

AgeLast
18

MAX(): The MAX() function returns the maximum value of the selected column.

Syntax:

```
SELECT MAX(column_name) FROM table_name;
```

Queries:

Fetching maximum marks among students from the Students table.

```
SELECT MAX(MARKS) AS MaxMarks FROM Students;
```

Output:

MaxMarks
95

- Fetching max age among students from the Students table.

```
SELECT MAX(AGE) AS MaxAge FROM Students;
```

Output:

MaxAge
21

MIN(): The MIN() function returns the minimum value of the selected column.

Syntax:

```
SELECT MIN(column_name) FROM table_name;
```

Queries:

- Fetching minimum marks among students from the Students table.

```
SELECT MIN(MARKS) AS MinMarks FROM Students;
```

Output:

MinMarks
50

- Fetching minimum age among students from the Students table.

```
SELECT MIN(AGE) AS MinAge FROM Students;
```

Output:

MinAge
18

SUM(): The SUM() function returns the sum of all the values of the selected column.

Syntax:

```
SELECT SUM(column_name) FROM table_name;
```

Queries:

- Fetching summation of total marks among students from the Students table.

```
SELECT SUM(MARKS) AS TotalMarks FROM Students;
```

Output:

TotalMarks
400

- Fetching summation of total age among students from the Students table.

```
SELECT SUM(AGE) AS TotalAge FROM Students;
```

Output:

TotalAge
97

Scalar Functions

UCASE(): It converts the value of a field to uppercase.

Syntax:

```
SELECT UCASE(column_name) FROM table_name;
```

Queries:

Converting names of students from the table Students to uppercase.

```
SELECT UCASE(NAME) FROM Students;
```

Output:

NAME
HARSH
SURESH
PRATIK
DHANRAJ
RAM

LCASE(): It converts the value of a field to lowercase.

Syntax:

```
SELECT LCASE(column_name) FROM table_name;
```

Queries:

Converting names of students from the table Students to lowercase.

```
SELECT LCASE(NAME) FROM Students;
```

Output:

NAME
harsh
suresh
pratik
dhanraj
ram

MID(): The MID() function extracts texts from the text field.

Syntax:

```
SELECT MID(column_name,start,length) AS some_name F
ROM table_name;

specifying length is optional here, and start signi
fies start position ( starting from 1 )
```

Queries:

- Fetching first four characters of names of students from the Students table.

```
SELECT MID(NAME,1,4) FROM Students;
```

Output:

NAME
HARS
SURE

PRAT
DHAN
RAM

LEN(): The LEN() function returns the length of the value in a text field.

Syntax:

```
SELECT LENGTH(column_name) FROM table_name;
```

Queries:

- Fetching length of names of students from Students table.

```
SELECT LENGTH(NAME) FROM Students;
```

Output:

NAME
5
6
6
7
3

ROUND(): The ROUND() function is used to round a numeric field to the number of decimals specified.NOTE: Many database systems have adopted the IEEE 754 standard for arithmetic operations, which says that when any numeric .5 is rounded it results to the nearest even integer i.e, 5.5 and 6.5 both gets rounded off to 6.

Syntax:

```
SELECT ROUND(column_name,decimals) FROM table_name;
decimals- number of decimals to be fetched.
```

Queries:

- Fetching maximum marks among students from the Students table.

```
SELECT ROUND(MARKS,0) FROM table_name;
```

Output:

MARKS
90
50
80
95
85

NOW(): The NOW() function returns the current system date and time.

Syntax:

```
SELECT NOW() FROM table_name;
```

Queries:

- Fetching current system time.

```
SELECT NAME, NOW() AS DateTime FROM Students;
```

Output:

NAME	DateTime
HARSH	1/13/2017 1:30:11 PM
SURESH	1/13/2017 1:30:11 PM
PRATIK	1/13/2017 1:30:11 PM
DHANRAJ	1/13/2017 1:30:11 PM
RAM	1/13/2017 1:30:11 PM

FORMAT(): The FORMAT() function is used to format how a field is to be displayed.

Syntax:

```
SELECT FORMAT(column_name,format) FROM table_name;
```

Queries:

- Formatting current date as 'YYYY-MM-DD'.

```
SELECT NAME, FORMAT(Now(),'YYYY-MM-DD') AS Date FRO
M Students;
```

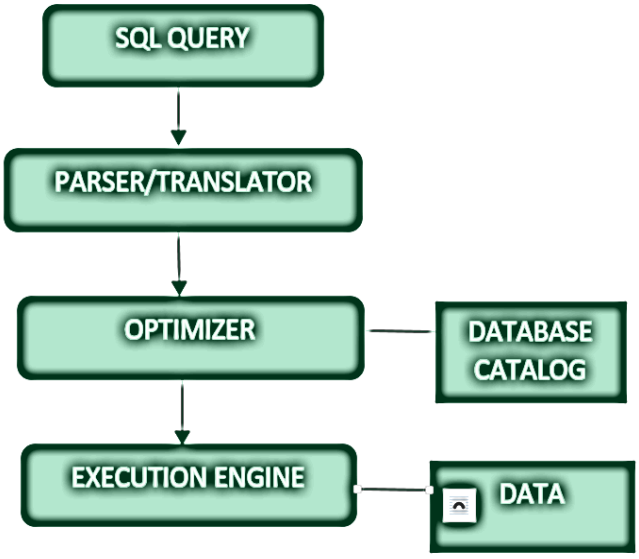
Output:

NAME	Date
HARSH	2017-01-13
SURESH	2017-01-13
PRATIK	2017-01-13
DHANRAJ	2017-01-13
RAM	2017-01-13

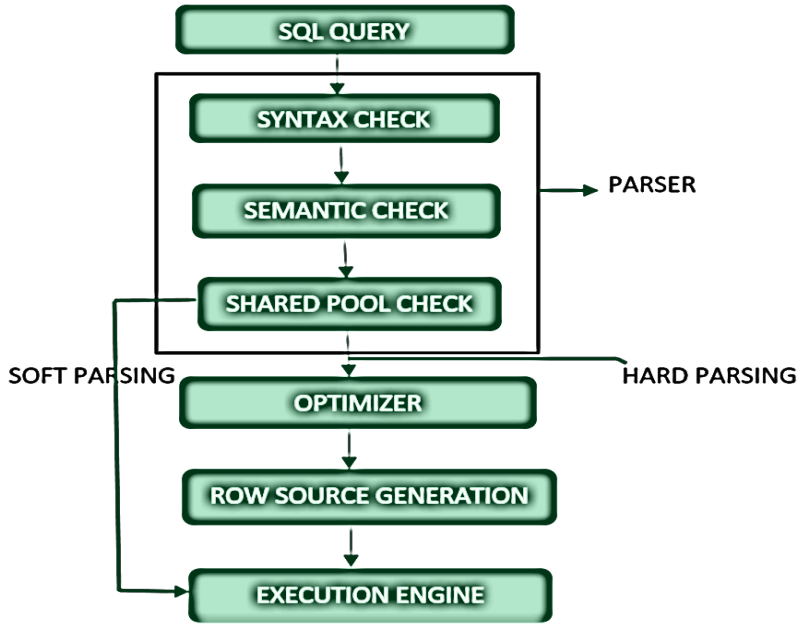
Query Processing-

Query Processing includes translations on high level Queries into low level expressions that can be used at physical level of file system, query optimization and actual execution of query to get the actual result.

Block Diagram of Query Processing is as:



Detailed Diagram is drawn as:



It is done in the following steps:

- **Step-1:**
  Parser: During parse call, the database performs the following checks- Syntax check, Semantic check and Shared pool check, after converting the query into relational algebra.

  Parser performs the following checks as (refer detailed diagram):

  Syntax check - concludes SQL syntactic validity.
  Example:

```
SELECT * FORM employee
```

Here error of wrong spelling of FROM is given by this check.

Semantic check - determines whether the statement is meaningful or not. Example: query contains a tablename which does not exist is checked by this check.

Shared Pool check - Every query possess a hash code during its execution. So, this check determines existence of written hash code in shared pool if code exists in shared pool then database will not take additional steps for optimization and execution.

Hard Parse and Soft Parse -

If there is a fresh query and its hash code does not exist in shared pool then that query has to pass through from the additional steps known as hard parsing otherwise if hash code exists then query does not passes through additional steps. It just passes directly to execution engine (refer detailed diagram). This is known as soft parsing.
Hard Parse includes following steps - Optimizer and Row source generation.

- **Step-2:**
  Optimizer: During optimization stage, database must perform a hard parse atleast for one unique DML statement and perform optimization during this parse. This database never optimizes DDL unless it includes a DML component such as subquery that require optimization.

It is a process in which multiple query execution plan for satisfying a query are examined and most efficient query plan is satisfied for execution.
Database catalog stores the execution plans and then optimizer passes the lowest cost plan for execution.

Row Source Generation -
The Row Source Generation is a software that receives a optimal execution plan from the optimizer and produces an iterative execution plan that is usable by the rest of the database. the iterative plan is the binary program that when executes by the sql engine produces the result set.

- **Step-3:**
  Execution Engine: Finally runs the query and display the required result.

WHERE Clause-

WHERE keyword is used for fetching **filtered data** in a result set.

- It is used to fetch data according to a particular criteria.
- WHERE keyword can also be used to filter data by matching patterns.

Basic Syntax: SELECT column1,column2 FROM table_name WHERE column_name operator value;

```
column1 , column2: fields int the table
table_name: name of table
column_name: name of field used for filtering the data
```

```
operator: operation to be considered for filtering
value: exact value or pattern to get related data i
n result
```

List of operators that can be used with where clause:

operator	description
>	Greater Than
>=	Greater than or Equal to
<	Less Than
<=	Less than or Equal to
=	Equal to
<>	Not Equal to
BETWEEN	In an inclusive Range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

Queries

- To fetch record of students with age equal to 20

```
SELECT * FROM Student WHERE Age=20;
```

Output:

ROLL_NO	NAME	ADDRESS	PHONE	Age
3	SUJIT	ROHTAK	XXXXXXXXXX	20
3	SUJIT	ROHTAK	XXXXXXXXXX	20

- To fetch Name and Address of students with ROLL_NO greater than 3

```
SELECT ROLL_NO,NAME,ADDRESS FROM Student WHERE ROLL
_NO > 3;
```

Output:

ROLL_NO	NAME	ADDRESS
4	SURESH	Delhi

BETWEEN operator

It is used to fetch filtered data in a given range inclusive of two values. **Basic Syntax: SELECT column1,column2 FROM table_name WHERE column_name BETWEEN value1 AND value2;**

```
BETWEEN: operator name
```

value1 AND value2: exact value from value1 to value2 to get related data in result set.

Queries

- To fetch records of students where ROLL_NO is between 1 and 3 (inclusive)

```
SELECT * FROM Student WHERE ROLL_NO BETWEEN 1 AND 3
;
```

Output:

ROLL_NO	NAME	ADDRESS	PHONE	Age
1	Ram	Delhi	XXXXXXXXXX	18
2	RAMESH	GURGAON	XXXXXXXXXX	18
3	SUJIT	ROHTAK	XXXXXXXXXX	20
3	SUJIT	ROHTAK	XXXXXXXXXX	20
2	RAMESH	GURGAON	XXXXXXXXXX	18

- To fetch NAME,ADDRESS of students where Age is between 20 and 30 (inclusive)

```
SELECT NAME,ADDRESS FROM Student WHERE Age BETWEEN
20 AND 30;
```

Output:

NAME	ADDRESS
SUJIT	Rohtak
SUJIT	Rohtak

LIKE operator

It is used to fetch filtered data by searching for a particular pattern in where clause. **Basic Syntax: SELECT column1,column2 FROM table_name WHERE column_name LIKE pattern;**

```
LIKE: operator name
```

pattern: exact value extracted from the pattern to get related data in result set. **Note**: The character(s) in pattern are case sensitive.

Queries

To fetch records of students where NAME starts with letter S.

```
SELECT * FROM Student WHERE NAME LIKE 'S%';
```

- The '%'(wildcard) signifies the later characters here which can be of any length and value.More about wildcards will be discussed in the later set. Output:

ROLL_NO	NAME	ADDRESS	PHONE	Age
3	SUJIT	ROHTAK	XXXXXXXXXX	20
4	SURESH	Delhi	XXXXXXXXXX	18
3	SUJIT	ROHTAK	XXXXXXXXXX	20

To fetch records of students where NAME contains the pattern 'AM'.

```
SELECT * FROM Student WHERE NAME LIKE '%AM%';
```

Output:

ROLL_NO	NAME	ADDRESS	PHONE	Age
1	Ram	Delhi	XXXXXXXXXX	18
2	RAMESH	GURGAON	XXXXXXXXXX	18
2	RAMESH	GURGAON	XXXXXXXXXX	18

IN operator

It is used to fetch filtered data same as fetched by '=' operator just the difference is that here we can specify multiple values for which we can get the result set. **Basic Syntax: SELECT column1,column2 FROM table_name WHERE column_name IN (value1,value2,..);**

```
IN: operator name
```

value1,value2,..: exact value matching the values given and get related data in result set.

Queries

- To fetch NAME and ADDRESS of students where Age is 18 or 20.

```
SELECT NAME,ADDRESS FROM Student WHERE Age IN (18,20);
```

Output:

NAME	ADDRESS
Ram	Delhi
RAMESH	GURGAON
SUJIT	ROHTAK
SURESH	Delhi
SUJIT	ROHTAK
RAMESH	GURGAON

To fetch records of students where ROLL_NO is 1 or 4.

```
SELECT * FROM Student WHERE ROLL_NO IN (1,4);
```

Output:

ROLL_NO	NAME	ADDRESS	PHONE	Age
1	Ram	Delhi	XXXXXXXXXX	18
4	SURESH	Delhi	XXXXXXXXXX	18

HIMANSHU KUMAR(LINKEDIN)

https://www.linkedin.com/in/himanshukumarmahuri

CREDITS- INTERNET

DISCLOSURE- THE DATA AND IMAGES ARE TAKEN FROM GOOGLE AND INTERNET.

CHECKOUT AND DOWNLOAD MY ALL NOTES

***LINK*- https://linktr.ee/exclusive_notes**