

UNIT-2

Relational Algebra

- Relational Algebra is procedural query language, which takes Relation as input and generates relation as output. Relational algebra mainly provides theoretical foundation for relational databases and SQL.
- Relational algebra is a procedural query language, it means that it tells what data to be retrieved and how to be retrieved.
- Relational Algebra works on the whole table at once, so we do not have to use loops etc to iterate over all the rows (tuples) of data one by one.
- All we have to do is specify the table name from which we need the data, and in a single line of command, relational algebra will traverse the entire given table to fetch data for you.

We can use Relational Algebra to fetch data from this Table(relation)

Select Name students with age less than 17

Output

Name
Ckon
Dkon

ID	Name	Age
1	Akon	17
2	Bkon	19
3	Ckon	15
4	Dkon	13

The output for query is also in form of a table(relation), with results in different columns

Basic/Fundamental Operations:

1. Select (σ)
2. Project (Π)
3. Union (\cup)
4. Set Difference ($-$)
5. Cartesian product (\times)
6. Rename (ρ)

1. Select Operation (σ) : This is used to fetch rows (tuples) from table(relation) which satisfies a given condition.

Syntax: $\sigma_p(r)$

- σ is the predicate
- r stands for relation which is the name of the table
- p is propositional logic

ex: $\sigma_{age > 17}(\text{Student})$

This will fetch the tuples(rows) from table **Student**, for which **age** will be greater than **17**.

$\sigma_{age > 17 \text{ and gender} = \text{'Male'}}(\text{Student})$

This will return tuples(rows) from table **Student** with information of male students, of age more than 17.

BRANCH_NAME	LOAN_NO	AMOUNT
Downtown	L-17	1000
Redwood	L-23	2000
Perryride	L-15	1500
Downtown	L-14	1500
Mianus	L-13	500
Roundhill	L-11	900
Perryride	L-16	1300

Input:

σ BRANCH_NAME="perryride" (LOAN)

Output:

BRANCH_NAME	LOAN_NO	AMOUNT
Perryride	L-15	1500
Perryride	L-16	1300

Project Operation (Π):

- Project operation is used to project only a certain set of attributes of a relation. In simple words, If you want to see only the **names** all of the students in the **Student** table, then you can use Project Operation.
- It will only project or show the columns or attributes asked for, and will also remove duplicate data from the columns.

Syntax of Project Operator (Π)

Π column_name1, column_name2, ..., column_nameN(table_name)

Example:

Π Name, Age(Student)

Above statement will show us only the **Name** and **Age** columns for all the rows of data in **Student** table.

Example: CUSTOMER RELATION

NAME	STREET	CITY
Jones	Main	Harrison
Smith	North	Rye
Hays	Main	Harrison

Curry	North	Rye
Johnson	Alma	Brooklyn
Brooks	Senator	Brooklyn

Input:

Π NAME, CITY (CUSTOMER)

Output:

NAME	CITY
Jones	Harrison
Smith	Rye
Hays	Harrison
Curry	Rye
Johnson	Brooklyn
Brooks	Brooklyn

Union Operation (\cup):

- This operation is used to fetch data from two relations(tables) or temporary relation(result of another operation).
- For this operation to work, the relations(tables) specified should have same number of attributes(columns) and same attribute domain. Also the duplicate tuples are automatically eliminated from the result.

Syntax: $A \cup B$

$\Pi_{\text{Student}}(\text{RegularClass}) \cup \Pi_{\text{Student}}(\text{ExtraClass})$

Example:

DEPOSITOR RELATION

CUSTOMER_NAME	ACCOUNT_NO
Johnson	A-101
Smith	A-121
Mayes	A-321
Turner	A-176
Johnson	A-273
Jones	A-472
Lindsay	A-284

BORROW RELATION

CUSTOMER_NAME	LOAN_NO
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11

Williams	L-17
----------	------

Input: $\Pi \text{ CUSTOMER_NAME (BORROW)} \cup \Pi \text{ CUSTOMER_NAME (DEPOSITOR)}$ **Output:**

CUSTOMER_NAME
Johnson
Smith
Hayes
Turner
Jones
Lindsay
Jackson
Curry
Williams
Mayes

Set Difference (-):

This operation is used to find data present in one relation and not present in the second relation. This operation is also applicable on two relations, just like Union operation.

Syntax: $A - B$

where A and B are relations.

For example, if we want to find name of students who attend the regular class but not the extra class, then, we can use the below operation:

 $\Pi_{\text{Student}}(\text{RegularClass}) - \Pi_{\text{Student}}(\text{ExtraClass})$

Input: $\prod \text{CUSTOMER_NAME (BORROW)} \cap \prod \text{CUSTOMER_NAME (DEPOSITOR)}$

CUSTOMER_NAME
Smith
Jones

Cartesian Product (X):

This is used to combine data from two different relations (tables) into one and fetch data from the combined relation.

Syntax: A X B

For example, if we want to find the information for Regular Class and Extra Class which are conducted during morning, then, we can use the following operation:

$\sigma_{\text{time} = \text{'morning'}} (\text{RegularClass X ExtraClass})$

For the above query to work, both **RegularClass** and **ExtraClass** should have the attribute **time**.

Notation: E X D

EMPLOYEE

EMP_ID	EMP_NAME	EMP_DEPT
1	Smith	A
2	Harry	C
3	John	B



DEPARTMENT

DEPT_NO	DEPT_NAME
A	Marketing
B	Sales
C	Legal

Input:

EMPLOYEE X DEPARTMENT

Output:

EMP_ID	EMP_NAME	EMP_DEPT	DEPT_NO	DEPT_NAME
1	Smith	A	A	Marketing
1	Smith	A	B	Sales
1	Smith	A	C	Legal
2	Harry	C	A	Marketing
2	Harry	C	B	Sales
2	Harry	C	C	Legal
3	John	B	A	Marketing
3	John	B	B	Sales



3	John	B	C	Legal
---	------	---	---	-------

Rename Operation (ρ):

This operation is used to rename the output relation for any query operation which returns result like Select, Project etc. Or to simply rename a relation(table)

Syntax: $\rho(\text{RelationNew}, \text{RelationOld})$

The rename operation is used to rename the output relation. It is denoted by **ρ** (ρ).

Example: We can use the rename operator to rename STUDENT relation to STUDENT1.
 $\rho(\text{STUDENT1}, \text{STUDENT})$

Join in DBMS:

- A JOIN clause is used to combine rows from two or more tables, based on a related column between them.
- **Join in DBMS** is a binary operation which allows you to combine join product and selection in one single statement.
- The goal of creating a join condition is that it helps you to combine the data from two or more DBMS tables.
- The tables in DBMS are associated using the primary key and foreign keys.

Types of SQL JOIN

1. INNER JOIN
2. LEFT JOIN
3. RIGHT JOIN
4. FULL JOIN

Table name: EMPLOYEE

EMP_ID	EMP_NAME	CITY	SALARY	AGE
1	Angelina	Chicago	200000	30
2	Robert	Austin	300000	26
3	Christian	Denver	100000	42
4	Kristen	Washington	500000	29
5	Russell	Los angels	200000	36
6	Marry	Canada	600000	48

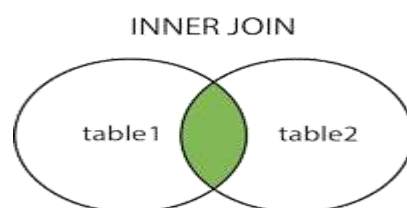
PROJECT

PROJECT_NO	EMP_ID	DEPARTMENT
101	1	Testing
102	2	Development
103	3	Designing
104	4	Development

1. INNER JOIN

In SQL, INNER JOIN selects records that have matching values in both tables as long as the condition is satisfied.

It returns the combination of all rows from both the tables where the condition satisfies.



Syntax

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

Query

```
SELECT EMPLOYEE.EMP_NAME, PROJECT.DEPARTMENT  
FROM EMPLOYEE INNER JOIN PROJECT  
ON PROJECT.EMP_ID = EMPLOYEE.EMP_ID;
```

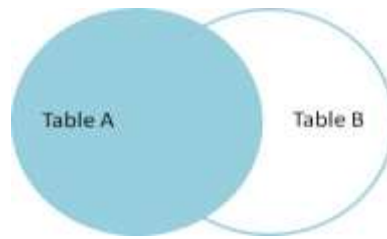
Output

EMP_NAME	DEPARTMENT
Angelina	Testing
Robert	Development
Christian	Designing
Kristen	Development

2. LEFT JOIN

The SQL left join returns all the values from left table and the matching values from the right table. If there is no matching join value, it will return NULL.





Syntax

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

Query

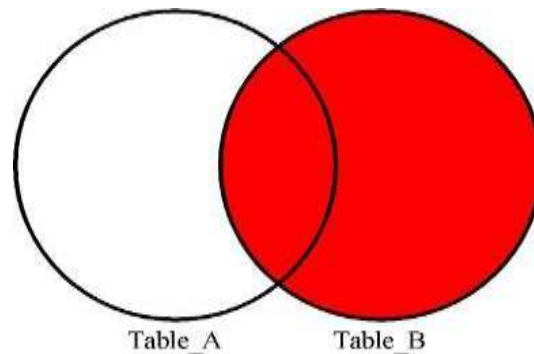
```
SELECT EMPLOYEE.EMP_NAME, PROJECT.DEPARTMENT  
FROM EMPLOYEE LEFT JOIN PROJECT  
ON PROJECT.EMP_ID = EMPLOYEE.EMP_ID;
```

Output

EMP_NAME	DEPARTMENT
Angelina	Testing
Robert	Development
Christian	Designing
Kristen	Development
Russell	NULL
Marry	NULL

3. RIGHT JOIN

In SQL, RIGHT JOIN returns all the values from the values from the rows of right table and the matched values from the left table. If there is no matching in both tables, it will return NULL.



Syntax

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

Query

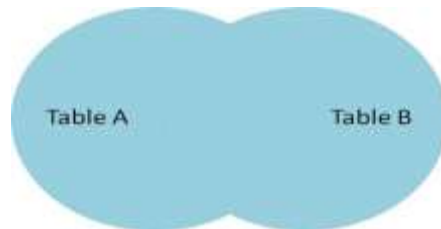
```
SELECT EMPLOYEE.EMP_NAME, PROJECT.DEPARTMENT
FROM EMPLOYEE RIGHT JOIN PROJECT
ON PROJECT.EMP_ID = EMPLOYEE.EMP_ID;
```

Output

EMP_NAME	DEPARTMENT
Angelina	Testing
Robert	Development
Christian	Designing
Kristen	Development

4. FULL JOIN

In SQL, FULL JOIN is the result of a combination of both left and right outer join. Join tables have all the records from both tables. It puts NULL on the place of matches not found.



Syntax

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

Query

```
SELECT EMPLOYEE.EMP_NAME, PROJECT.DEPARTMENT
FROM EMPLOYEE
FULL JOIN PROJECT
ON PROJECT.EMP_ID = EMPLOYEE.EMP_ID;
```

Output

EMP_NAME	DEPARTMENT
Angelina	Testing
Robert	Development
Christian	Designing
Kristen	Development
Russell	NULL



Marry	NULL
-------	------

Division Operator in SQL

Division Operator (\div): Division operator $A \div B$ can be applied if and only if:

- Attributes of B is proper subset of Attributes of A.
- The relation returned by division operator will have attributes = (All attributes of A – All Attributes of B)
- The relation returned by division operator will return those tuples from relation A which are associated to every B's tuple.

The division operator is used when we have to evaluate queries which contain the keyword **ALL**.

A		B1	B2	B3
sno	pno	pno	pno	pno
s1	p1	p2	p2	p1
s1	p2		p4	p2
s1	p3			p4
s1	p4			
s2	p1			
s2	p2			
s3	p2			
s4	p2			
s4	p4			

A/B1		A/B2		A/B3
sno		sno		sno
s1		s1		s1
s2		s2		
s3		s3		
s4		s4		

Table 1: Course_Taken → It consists of the names of Students against the courses that they have taken.

Student_Name	Course
Robert	Databases

Robert	Programming Languages
David	Databases
David	Operating Systems
Hannah	Programming Languages
Hannah	Machine Learning
Tom	Operating Systems

Table 2: Course_Required → It consists of the courses that one is required to take in order to graduate.

Course
Databases
Programming Languages

1. Find all the students

Create a set of all students that have taken courses. This can be done easily using the following command.

CREATE TABLE AllStudents AS SELECT DISTINCT Student_Name FROM Course_Taken

This command will return the table **AllStudents**, as the resultset:

Student_name
Robert
David
Hannah
Tom

2. Find all the students and the courses required to graduate

Next, we will create a set of students and the courses they need to graduate. We can express this in the form of Cartesian Product of **AllStudents** and **Course_Required** using the following command.

**CREATE table StudentsAndRequired AS
SELECT AllStudents.Student_Name, Course_Required.Course
FROM AllStudents, Course_Required**

Now the new resultset - table **StudentsAndRequired** will be:

Student_Name	Course
Robert	Databases
Robert	Programming Languages
David	Databases
David	Programming Languages
Hannah	Databases
Hannah	Programming Languages
Tom	Databases
Tom	Programming Languages

Relational Calculus:

Relational calculus is a non-procedural query language that tells the system what data to be retrieved but doesn't tell how to retrieve it. Relational Calculus exists in two forms:

1. Tuple Relational Calculus (TRC)
2. Domain Relational Calculus (DRC)

Tuple Relational Calculus (TRC)

Tuple relational calculus is used for selecting those tuples that satisfy the given condition.

Table: Student

First_Name	Last_Name	Age

Ajeet	Singh	30
Chaitanya	Singh	31
Rajeev	Bhatia	27
Carl	Pratap	28

Lets write relational calculus queries.

Query to display the last name of those students where age is greater than 30

```
{ t.Last_Name | Student(t) AND t.age > 30 }
```

In the above query you can see two parts separated by | symbol. The second part is where we define the condition and in the first part we specify the fields which we want to display for the selected tuples.

The result of the above query would be:

```
Last_Name
-----
Singh
```

Query to display all the details of students where Last name is 'Singh'

```
{ t | Student(t) AND t.Last_Name = 'Singh' }
```

Output:

First_Name	Last_Name	Age
-----	-----	----
Ajeet	Singh	30
Chaitanya	Singh	31

Ex:

Table-1: Customer

Customer name	Street	City
Saurabh	A7	Patiala
Mehak	B6	Jalandhar
Sumiti	D9	Ludhiana
Ria	A5	Patiala

Table-2: Branch

Branch name	Branch city
ABC	Patiala
DEF	Ludhiana
GHI	Jalandhar

Table-3: Account

Account number	Branch name	Balance
1111	ABC	50000
1112	DEF	10000
1113	GHI	9000



Account number	Branch name	Balance
1114	ABC	7000

Table-4: Loan

Loan number	Branch name	Amount
L33	ABC	10000
L35	DEF	15000
L49	GHI	9000
L98	DEF	65000

Table-5: Borrower

Customer name	Loan number
Saurabh	L33
Mehak	L49
Ria	L98

Table-6: Depositor

Customer name	Account number
Saurabh	1111

Customer name	Account number
Mehak	1113
Sumiti	1114

Queries-1: Find the loan number, branch, amount of loans of greater than or equal to 10000 amount.

$\{t \mid t \in \text{loan} \wedge t[\text{amount}] \geq 10000\}$

Resulting relation:

Loan number	Branch name	Amount
L33	ABC	10000
L35	DEF	15000
L98	DEF	65000

Domain Relational Calculus (DRC)

In domain relational calculus the records are filtered based on the domains.

Again we take the same table to understand how DRC works.

Table: Student

First_Name	Last_Name	Age

Ajeet	Singh	30
Chaitanya	Singh	31
Rajeev	Bhatia	27
Carl	Pratap	28

Query to find the first name and age of students where student age is greater than 27

$\{ \langle \text{First_Name}, \text{Age} \rangle \mid \langle \text{First_Name}, \text{Age} \rangle \in \text{Student} \wedge \text{Age} > 27 \}$

Note:

The symbols used for logical operators are: \wedge for AND, \vee for OR and \neg for NOT.

Output:

First_Name	Age
Ajeet	30
Chaitanya	31
Carl	28

SQL Basic Structure

1. Basic structure of an SQL expression consists of **select**, **from** and **where** clauses.
 - **select** clause lists attributes to be copied - corresponds to relational algebra **project**.
 - **from** clause corresponds to Cartesian product - lists relations to be used.
 - **where** clause corresponds to selection predicate in relational algebra.

The SELECT statement is used to select data from a database.

The data returned is stored in a result table, called the result-set.

To fetch the entire table or all the fields in the table:

```
SELECT * FROM table_name;
```

To fetch individual column data

```
SELECT column1,column2 FROM table_name
```

WHERE SQL clause

WHERE clause is used to specify/apply any condition while retrieving, updating or deleting data from a table. This clause is used mostly with **SELECT**, **UPDATE** and **DELETE** query.

The basic syntax of the SELECT statement with the WHERE clause is as shown below.

```
SELECT column1, column2, columnN
```

```
FROM table_name
```

WHERE [condition]

Example

Consider the CUSTOMERS table having the following records –

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

The following code is an example which would fetch the ID, Name and Salary fields from the CUSTOMERS table, where the salary is greater than 2000 –

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000;
```

This would produce the following result –

ID	NAME	SALARY
4	Chaitali	6500.00
5	Hardik	8500.00
6	Komal	4500.00
7	Muffy	10000.00



From clause:

From clause can be used to specify a sub-query expression in SQL. The relation produced by the sub-query is then used as a new relation on which the outer query is applied.

- Sub queries in the from clause are supported by most of the SQL implementations.
- The correlation variables from the relations in from clause cannot be used in the sub-queries in the from clause.

Syntax:

SELECT column1, column2 FROM

(SELECT column_x as C1, column_y FROM table WHERE PREDICATE_X)

as table2

WHERE PREDICATE;

SET Operations

SQL supports few Set operations which can be performed on the table data. These are used to get meaningful results from data stored in the table, under different special conditions.

In this tutorial, we will cover 4 different types of SET operations, along with example:

1. UNION
2. UNION ALL
3. INTERSECT
4. MINUS

1. Union

- The SQL Union operation is used to combine the result of two or more SQL SELECT queries.
 - In the union operation, all the number of datatype and columns must be same in both the tables on which UNION operation is being applied.
 - The union operation eliminates the duplicate rows from its resultset.
-

Syntax

SELECT column_name FROM table1

UNION

SELECT column_name FROM table2;

The First table

ID	NAME
1	Jack
2	Harry
3	Jackson

The Second table

ID	NAME
3	Jackson
4	Stephan
5	David

Union SQL query will be:

SELECT * FROM First

UNION

SELECT * FROM Second;

The resultset table will look like:

ID	NAME
1	Jack
2	Harry
3	Jackson
4	Stephan
5	David

2. Union All

Union All operation is equal to the Union operation. It returns the set without removing duplication and sorting the data.

Syntax:

```
SELECT column_name FROM table1
UNION ALL
SELECT column_name FROM table2;
```

Example: Using the above First and Second table.

Union All query will be like:

```
SELECT * FROM First
UNION ALL
SELECT * FROM Second;
```

The resultset table will look like:



-ID	NAME
1	Jack
2	Harry
3	Jackson
3	Jackson
4	Stephan
5	David

3. Intersect

- It is used to combine two SELECT statements. The Intersect operation returns the common rows from both the SELECT statements.
- In the Intersect operation, the number of datatype and columns must be the same.
- It has no duplicates and it arranges the data in ascending order by default.

Syntax

```
SELECT column_name FROM table1
INTERSECT
SELECT column_name FROM table2;
```

Example:

Using the above First and Second table.

Intersect query will be:

```
SELECT * FROM First
INTERSECT
SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
3	Jackson

4. Minus

- It combines the result of two SELECT statements. Minus operator is used to display the rows which are present in the first query but absent in the second query.
- It has no duplicates and data arranged in ascending order by default.

Syntax:

```
SELECT column_name FROM table1
MINUS
SELECT column_name FROM table2;
```

Example

Using the above First and Second table.

Minus query will be:

```
SELECT * FROM First
MINUS
SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
1	Jack
2	Harry

Aggregate functions in SQL

- SQL aggregation function is used to perform the calculations on multiple rows of a single column of a table. It returns a single value.
- It is also used to summarize the data.

Aggregate Functions

- 1) Count()
- 2) Sum()
- 3) Avg()
- 4) Min()
- 5) Max()

1. COUNT FUNCTION

- COUNT function is used to Count the number of rows in a database table. It can work on both numeric and non-numeric data types.
- COUNT function uses the COUNT(*) that returns the count of all the rows in a specified table. COUNT(*) considers duplicate and Null.

Count(*): Returns total number of records



PRODUCT_MAST

PRODUCT	COMPANY	QTY	RATE	COST
Item1	Com1	2	10	20
Item2	Com2	3	25	75
Item3	Com1	2	30	60
Item4	Com3	5	10	50
Item5	Com2	2	20	40
Item6	Cpm1	3	25	75
Item7	Com1	5	30	150
Item8	Com1	3	10	30
Item9	Com2	2	25	50
Item10	Com3	4	30	120

Example: COUNT()

```
SELECT COUNT(*) FROM PRODUCT_MAST;
```

Output:

```
10
```

Example: COUNT with WHERE

```
SELECT COUNT(*)  
FROM PRODUCT_MAST;  
WHERE RATE>=20;
```

Output:7

Example: COUNT() with DISTINCT

```
SELECT COUNT(DISTINCT COMPANY)
FROM PRODUCT_MAST;
```

Output:

```
3
```

2. SUM Function

Sum function is used to calculate the sum of all selected columns. It works on numeric fields only.

Syntax

```
SUM()
```

or

```
SUM( [ALL|DISTINCT] expression )
```

Example: SUM()

```
SELECT SUM(COST)
FROM PRODUCT_MAST;
```

Output:

```
670
```

Example: SUM() with WHERE

```
SELECT SUM(COST)
FROM PRODUCT_MAST
WHERE QTY>3;
```

Output:

```
320
```

3. AVG function

The AVG function is used to calculate the average value of the numeric type. AVG function returns the average of all non-Null values.

Syntax

AVG()

Example:

```
SELECT AVG(COST)
FROM PRODUCT_MAST;
```

Output:

67.00

4. MAX Function

MAX function is used to find the maximum value of a certain column. This function determines the largest value of all selected values of a column.

Syntax: MAX()**Example:**

```
SELECT MAX(RATE)
FROM PRODUCT_MAST;
```

30

5. MIN Function

MIN function is used to find the minimum value of a certain column. This function determines the smallest value of all selected values of a column.

Syntax: MIN())**Example:** SELECT MIN(RATE)

```
FROM PRODUCT_MAST;
```

Output:10**GROUP BY Statement**

The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

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

GROUP BY Syntax

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

SI NO	NAME	SALARY	AGE	SUBJECT	YEAR	NAME
1	Harsh	2000	19	English	1	Harsh
2	Dhanraj	3000	20	English	1	Pratik
3	Ashish	1500	19	English	1	Ramesh
4	Harsh	3500	19	English	2	Ashish
5	Ashish	1500	19	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

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;
```

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

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;
```

Example

Consider the CUSTOMERS table having the following records.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Following is an example, which would display a record for a similar age count that would be more than or equal to 2.

```
SQL > SELECT ID, NAME, AGE, ADDRESS, SALARY
FROM CUSTOMERS
GROUP BY age
HAVING COUNT(age) >= 2;
```

This would produce the following result –

2	Khilan	25	Delhi	1500.00
---	--------	----	-------	---------

+.....+.....+.....+.....+.....+

Nested Queries

In nested queries, a query is written inside a query. The result of inner query is used in execution of outer query. We will use **STUDENT**, **COURSE**, **STUDENT_COURSE** tables for understanding nested queries.

STUDENT

S_ID	S_NAME	S_ADDRESS	S_PHONE	S_AGE
S1	RAM	DELHI	9455123451	18
S2	RAMESH	GURGAON	9652431543	18
S3	SUJIT	ROHTAK	9156253131	20
S4	SURESH	DELHI	9156768971	18

COURSE

C_ID	C_NAME
C1	DSA
C2	Programming
C3	DBMS

STUDENT_COURSE

S_ID	C_ID
------	------



S1	C1
S1	C3
S2	C1
S3	C2
S4	C2
S4	C3

Example

Consider the CUSTOMERS table having the following records –

```

+-----+-----+-----+-----+
| ID | NAME   | AGE | ADDRESS | SALARY |
+-----+-----+-----+-----+
| 1 | Ramesh | 35 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi     | 1500.00 |
| 3 | kaushik | 23 | Kota      | 2000.00 |
| 4 | Chaitali | 25 | Mumbai    | 6500.00 |
| 5 | Hardik | 27 | Bhopal     | 8500.00 |
| 6 | Komal | 22 | MP         | 4500.00 |
| 7 | Muffy | 24 | Indore     | 10000.00 |
+-----+-----+-----+-----+

```

Now, let us check the following subquery with a SELECT statement.

```

SQL> SELECT *
      FROM CUSTOMERS
      WHERE ID IN (SELECT ID

```

FROM CUSTOMERS

WHERE SALARY > 4500) ;

This would produce the following result.

```
+.....+.....+.....+.....+
| ID | NAME   | AGE | ADDRESS | SALARY |
+.....+.....+.....+.....+
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik  | 27 | Bhopal  | 8500.00 |
| 7 | Muffy   | 24 | Indore  | 10000.00 |
+.....+.....+.....+.....+
```

Students

id	name	class_id	GPA
1	Jack Black	3	3.45
2	Daniel White	1	3.15
3	Kathrine Star	1	3.85
4	Helen Bright	2	3.10
5	Steve May	2	2.40



Teachers

id	name	subject	class_id	monthly_salary
1	Elisabeth Grey	History	3	2,500
2	Robert Sun	Literature	[NULL]	2,000
3	John Churchill	English	1	2,350
4	Sara Parker	Math	2	3,000

Classes

id	grade	teacher_id	number_of_students
1	10	3	21
2	11	4	25
3	12	1	28

```
SELECT *  
FROM students  
WHERE GPA > (  
    SELECT AVG(GPA)  
    FROM students);
```

result:

id	name	class_id	GPA
1	Jack Black	3	3.45
3	Kathrine Star	1	3.85

```
SELECT AVG(number_of_students)
FROM classes
WHERE teacher_id IN (
    SELECT id
    FROM teachers
    WHERE subject = 'English' OR subject = 'History');
```

Views in SQL

- Views in SQL are considered as a virtual table. A view also contains rows and columns.
- To create the view, we can select the fields from one or more tables present in the database.
- A view can either have specific rows based on certain condition or all the rows of a table.

Sample table:

Student_Detail

STU_ID	NAME	ADDRESS
1	Stephan	Delhi

2	Kathrin	Noida
3	David	Ghaziabad
4	Alina	Gurugram

Student_Marks

STU_ID	NAME	MARKS	AGE
1	Stephan	97	19
2	Kathrin	86	21
3	David	74	18
4	Alina	90	20
5	John	96	18

1. Creating view

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

Syntax:

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE condition;
```



2. Creating View from a single table

Query:

```
CREATE VIEW DetailsView AS  
SELECT NAME, ADDRESS  
FROM Student_Details  
WHERE STU_ID < 4;
```

Just like table query, we can query the view to view the data.

```
SELECT * FROM DetailsView;
```

Output:

NAME	ADDRESS
Stephan	Delhi
Kathrin	Noida
David	Ghaziabad

3. Creating View from multiple tables

View from multiple tables can be created by simply include multiple tables in the SELECT statement.

In the given example, a view is created named MarksView from two tables Student_Detail and Student_Marks.

Query:

```
CREATE VIEW MarksView AS  
SELECT Student_Detail.NAME, Student_Detail.ADDRESS, Student_Marks.MARKS  
FROM Student_Detail, Student_Mark  
WHERE Student_Detail.NAME = Student_Marks.NAME;
```

To display data of View MarksView:

```
SELECT * FROM MarksView;
```

NAME	ADDRESS	MARKS
Stephan	Delhi	97
Kathrin	Noida	86
David	Ghaziabad	74
Alina	Gurugram	90

4. Deleting View

A view can be deleted using the Drop View statement.

Syntax

1. DROP VIEW view_name;

Example:

If we want to delete the View **MarksView**, we can do this as:

1. DROP VIEW MarksView;

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 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.

Trigger: A trigger is a stored procedure in database which automatically invokes whenever a special event in the database occurs. For example, a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated.

Syntax:

```
create trigger [trigger_name]
[before | after]
{insert | update | delete}
on [table_name]
[for each row]
[trigger_body]
```

Explanation of syntax:

1. create trigger [trigger_name]: Creates or replaces an existing trigger with the trigger_name.
 2. [before | after]: This specifies when the trigger will be executed.
 3. {insert | update | delete}: This specifies the DML operation.
 4. on [table_name]: This specifies the name of the table associated with the trigger.
 5. [for each row]: This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
 6. [trigger_body]: This provides the operation to be performed as trigger is fired
-

BEFORE and AFTER of Trigger:

BEFORE triggers run the trigger action before the triggering statement is run.

AFTER triggers run the trigger action after the triggering statement is run.

Example:

Given Student Report Database, in which student marks assessment is recorded. In such schema, create a trigger so that the total and average of specified marks is automatically inserted whenever a record is insert.

Here, as trigger will invoke before record is inserted so, BEFORE Tag can be used.

Suppose the database Schema –

```
mysql> desc Student;
```

Field	Type	Null	Key	Default	Extra
tid	int(4)	NO	PRI	NULL	auto_increment
name	varchar(30)	YES		NULL	
subj1	int(2)	YES		NULL	
subj2	int(2)	YES		NULL	
subj3	int(2)	YES		NULL	
total	int(3)	YES		NULL	
per	int(3)	YES		NULL	

7 rows in set (0.00 sec)

SQL Trigger to problem statement.

```
create trigger stud_marks
```

```
before INSERT
```

```
on
```

```
Student
```

```
for each row
```

```
set Student.total = Student.subj1 + Student.subj2 + Student.subj3, Student.per =  
Student.total * 60 / 100;
```

Above SQL statement will create a trigger in the student database in which whenever subjects marks are entered, before inserting this data into the database, trigger will compute those two values and insert with the entered values. i.e.,

```
mysql> insert into Student values(0, "ABCDE", 20, 20, 20, 0, 0);
```

Query OK, 1 row affected (0.09 sec)

```
mysql> select * from Student;
```

```
+-----+-----+-----+-----+-----+-----+
| tid | name | subj1 | subj2 | subj3 | total | per |
+-----+-----+-----+-----+-----+-----+
| 100 | ABCDE | 20 | 20 | 20 | 60 | 36 |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

In this way trigger can be created and executed in the databases.

Attention reader! Don't stop learning now. Get hold of all the important CS Theory concepts for SDE interviews with the [**CS Theory Course**](#) at a student-friendly price and become industry ready.

Advantages of Triggers

These are the following advantages of Triggers:

- Trigger generates some derived column values automatically
- Enforces referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Creating a trigger:

Syntax for creating trigger:

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{ BEFORE | AFTER | INSTEAD OF }
{ INSERT [OR] | UPDATE [OR] | DELETE }
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
```

[FOR EACH ROW]

WHEN (condition)

DECLARE

Declaration-statements

BEGIN

Executable-statements

EXCEPTION

Exception-handling-statements

END;

Here,

- **CREATE [OR REPLACE] TRIGGER trigger_name:** It creates or replaces an existing trigger with the trigger_name.
- **{ BEFORE | AFTER | INSTEAD OF } :** This specifies when the trigger would be executed. The INSTEAD OF clause is used for creating trigger on a view.
- **{ INSERT [OR] | UPDATE [OR] | DELETE }:** This specifies the DML operation.
- **[OF col_name]:** This specifies the column name that would be updated.
- **[ON table_name]:** This specifies the name of the table associated with the trigger.
- **[REFERENCING OLD AS o NEW AS n]:** This allows you to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.
- **[FOR EACH ROW]:** This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- **WHEN (condition):** This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

PL/SQL Trigger Example

Let's take a simple example to demonstrate the trigger. In this example, we are using the following CUSTOMERS table:

Create table and have records:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

Create trigger:

Let's take a program to create a row level trigger for the CUSTOMERS table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values:

```

CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
    sal_diff number;
BEGIN
    sal_diff := :NEW.salary - :OLD.salary;
    dbms_output.put_line('Old salary: ' || :OLD.salary);
    dbms_output.put_line('New salary: ' || :NEW.salary);
    dbms_output.put_line('Salary difference: ' || sal_diff);
END;

```

After the execution of the above code at SQL Prompt, it produces the following result.

Trigger created.

Check the salary difference by procedure:

Use the following code to get the old salary, new salary and salary difference after the trigger created.

DECLARE

```
total_rows number(2);
```

BEGIN

```
UPDATE customers
```

```
SET salary = salary + 5000;
```

```
IF sql%notfound THEN
```

```
    dbms_output.put_line('no customers updated');
```

```
ELSIF sql%found THEN
```

```
    total_rows := sql%rowcount;
```

```
    dbms_output.put_line( total_rows || ' customers updated ');
```

```
END IF;
```

```
END;
```

/ Output:

Old salary: 20000

New salary: 25000

Salary difference: 5000

Old salary: 22000

New salary: 27000

Salary difference: 5000

Old salary: 24000

New salary: 29000

Salary difference: 5000

Old salary: 26000

New salary: 31000

Salary difference: 5000

Old salary: 28000

New salary: 33000

Salary difference: 5000

```
Old salary: 30000
New salary: 35000
Salary difference: 5000
6 customers updated
```

Note: As many times you executed this code, the old and new both salary is incremented by 5000 and hence the salary difference is always 5000.

After the execution of above code again, you will get the following result.

```
Old salary: 25000
New salary: 30000
Salary difference: 5000
Old salary: 27000
New salary: 32000
Salary difference: 5000
Old salary: 29000
New salary: 34000
Salary difference: 5000
Old salary: 31000
New salary: 36000
Salary difference: 5000
Old salary: 33000
New salary: 38000
Salary difference: 5000
Old salary: 35000
New salary: 40000
Salary difference: 5000
6 customers updated
```

Important Points

Following are the two very important point and should be noted carefully.

- OLD and NEW references are used for record level triggers these are not avialable for table level triggers.



- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.

Procedure

The PL/SQL stored procedure or simply a procedure is a PL/SQL block which performs one or more specific tasks. It is just like procedures in other programming languages.

The procedure contains a header and a body.

- **Header:** The header contains the name of the procedure and the parameters or variables passed to the procedure.
- **Body:** The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

How to pass parameters in procedure:

When you want to create a procedure or function, you have to define parameters .There is three ways to pass parameters in procedure:

1. **IN parameters:** The IN parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or the function.
2. **OUT parameters:** The OUT parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
3. **INOUT parameters:** The INOUT parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

A procedure may or may not return any value.

PL/SQL Create Procedure

Syntax for creating procedure:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
    [ (parameter [,parameter]) ]
```

IS

```
[declaration_section]
BEGIN
    executable_section
[EXCEPTION
    exception_section]
END [procedure_name];
Create procedure example
```

In this example, we are going to insert record in user table. So you need to create user table first.

Table creation:

create table user(id number(10) **primary key**,name varchar2(100)); Now write the procedure code to insert record in user table.

Procedure Code:

```
create or replace procedure "INSERTUSER"
(id IN NUMBER,
name IN VARCHAR2)
is
begin
insert into user values(id,name);
end;
/
```

Output:

Procedure created.

PL/SQL program to call procedure

Let's see the code to call above created procedure.

```
BEGIN
    insertuser(101,'Rahul');
    dbms_output.put_line('record inserted successfully');
END;
/
```

Now, see the "USER" table, you will see one record is inserted.

ID	Name
101	Rahul

PL/SQL Drop Procedure

Syntax for drop procedure

DROP PROCEDURE procedure name;

Example of drop procedure

DROP PROCEDURE pro1;

