Relational Language & Relational Model

Introduction to SQL

- SQL(Structured Query Language)
- It is a database computer language designed for the retrieval and management of data in a relational database.
- It is a ANSI (American National Standard Institute) for relational database system.
- All the relational database management system like MySql, MSAccess, Oracle, Sybase and SQL Server use SQL as their standard database language.

RDBMS

It is a database management system based on the relational model introduced by E.F.Codd. In a relational model, data is stored in a relation (table) and is represented in form of tuples (rows). It is used to manage relational database. Relational database is a collection of organized tables related to each other and from which data can be accessed easily.

Tables

It is a collection of data elements organized in terms of rows and columns. Tables can have duplicate row of data but relation can't have duplicate row data.

Field

It is a column in a table that is designed to maintain specific information about every record in table.

Record of Row

A record or row is called as a row of data is each individual entry that exists in a table.

Column

A vertical entity in a table that contains all information associated with a specific field in a table.

Null Value

A Null Value in a table is a value in a field that appears to be blank, which means a field with null value is a field with no value. It is different than zero value.

Attribute

A table consists of several records(row), each record can be broken down into several smaller parts of data known as Attributes. So it is the properties of relation where its domain is defined to hold certain type of data values.

Relation Schema

A relation schema describes the structure of the relation, with the name of the relation(name of table), its attributes and their names and type.

Relation Key

A relation key is an attribute which can uniquely identify a particular tuple(row) in a relation(table).

Codd's Rule for Relational DBMS

E.F Codd was a Computer Scientist who invented the **Relational model** for Database management. Based on relational model, the **Relational database** was created. Codd proposed 13 rules popularly known as **Codd's 12 rules** to test DBMS's concept against his relational model. Codd's rule actually define what quality a DBMS requires in order to become a Relational Database Management System(RDBMS). Till now, there is hardly any commercial product that follows all the 13 Codd's rules. Even **Oracle** follows only eight and half(8.5) out of 13. The Codd's 12 rules are as follows.

Rule zero

This rule states that for a system to qualify as an **RDBMS**, it must be able to manage database entirely through the relational capabilities.

Rule 1: Information rule

All information(including metadata) is to be represented as stored data in cells of tables. The rows and columns have to be strictly unordered.

Rule 2: Guaranteed Access

Each unique piece of data(atomic value) should be accessible by : **Table Name + Primary Key(Row) + Attribute(column)**.

NOTE: Ability to directly access via POINTER is a violation of this rule.

Rule 3: Systematic treatment of NULL

Null has several meanings, it can mean missing data, not applicable or no value. It should be handled consistently. Also, Primary key must not be null, ever. Expression on NULL must give null.

Rule 4: Active Online Catalog

Database dictionary(catalog) is the structure description of the complete **Database** and it must be stored online. The Catalog must be governed by same rules as rest of the database. The same query language should be used on catalog as used to query database.

Rule 5: Powerful and Well-Structured Language

One well structured language must be there to provide all manners of access to the data stored in the database. Example: **SQL**, etc. If the database allows access to the data without the use of this language, then that is a violation.

Rule 6: View Update Rule

All the view that are theoretically updatable should be updatable by the system as well.

Rule 7: Relational Level Operation
There must be Insert, Delete, Update operations at each level of relations. Set operation like Union, Intersection and minus should also be supported.

Rule 8: Physical Data Independence
The physical storage of data should not matter to the system. If say, some file supporting table is renamed or moved from one disk to another, it should not affect the application.

Rule 9: Logical Data Independence
If there is change in the logical structure(table structures) of the database the user view of data should not change. Say, if a table is split into two tables, a new view should give result as the join of the two tables. This rule is most difficult to satisfy.

Rule 10: Integrity Independence
The database should be able to enforce its own integrity rather than using other programs. Key and Check constraints, trigger etc, should be stored in Data Dictionary. This also make **RDBMS** independent of front-end.

Rule 11: Distribution Independence
A database should work properly regardless of its distribution across a network. Even if a database is geographically distributed, with data stored in pieces, the end user should get an impression that it is stored at the same place. This lays the foundation of **distributed database**.

Rule 12: Non-subversion Rule
If low level access is allowed to a system it should not be able to subvert or bypass integrity rules to change the data. This can be achieved by some sort of looking or encryption.

Features of SQL

- High performance
- High availability
- Scalability and flexibility
- Robust Transactional Support
- High security
- Comprehensive Application development
- Open source
- Management ease

Queries & Sub-Queries

SQL Command(Queries)

SQL defines following ways to retrieve and manipulate data stored in an RDBMS.

Statements

DDL(Data Definition Language) : It includes changes to the structure of table like creation of tables, altering tables, deleting tables etc. All DDL commands are auto-committed. That means it saves all the changes permanently in the database.

Command	Description
create	to create new table or database
alter	for alteration
truncate	delete data from table
drop	to drop a table
rename	to rename a table

Create command

create is a DDL SQL command used to create a table or a database in relational database management system.

Creating a Database

To create a database in RDBMS, **create** command is used. Following is the syntax,

CREATE DATABASE <DB_NAME>;

Example for creating Database

CREATE DATABASE Test;

The above command will create a database named **Test**, which will be an empty schema without any table.

To create tables in this newly created database, we can again use the create command.

Creating a Table

create command can also be used to create tables. Now when we create a table, we have to specify the details of the columns of the tables too. We can specify the **names** and **datatypes** of various columns in the create command itself.

Following is the syntax,

```
CREATE TABLE <TABLE_NAME>
(
        column_name1 datatype1,
        column_name2 datatype2,
        column_name3 datatype3,
        column_name4 datatype4
);
```

create table command will tell the database system to create a new table with the given table name and column information.

Example for creating Table

```
CREATE TABLE Student(
                        student_id INT,
                        name VARCHAR(100),
                        age INT
                    );
```

ALTER command

alter command is used for altering the table structure, such as,

- to add a column to existing table
- to rename any existing column
- to change datatype of any column or to modify its size.
- to drop a column from the table.

ALTER Command: Add a new Column

Using ALTER command we can add a column to any existing table. Following is the syntax,

```
ALTER TABLE table_name ADD(column_name datatype);
```

Here is an Example for this,

```
ALTER TABLE student ADD(address VARCHAR(200));
```

The above command will add a new column address to the table **student**, which will hold data of type varchar which is nothing but string, of length 200.

ALTER Command: Add multiple new Columns

Using ALTER command we can even add multiple new columns to any existing table. Following is the syntax,

```
ALTER TABLE table_name ADD   column_name1 datatype1,
                             column-name2 datatype2,
                             column-name3 datatype3);
```

Here is an Example for this,

```
ALTER TABLE student ADD(
                             father_name VARCHAR(60),
                             mother_name VARCHAR(60),
                             dob DATE);
```

The above command will add three new columns to the **student** table

ALTER Command: Add Column with default value

ALTER command can add a new column to an existing table with a default value too. The default value is used when no value is inserted in the column. Following is the syntax,

```
ALTER TABLE table_name ADD(column-name1 datatype1 DEFAULT some_value);
```

Here is an Example for this,

ALTER TABLE student ADD(dob DATE DEFAULT '01-Jan-99');

The above command will add a new column with a preset default value to the table **student**.

ALTER Command: Modify an existing Column

ALTER command can also be used to modify data type of any existing column. Following is the syntax,

ALTER TABLE table_name modify(column_name datatype);

Here is an Example for this,

ALTER TABLE student MODIFY(address varchar(300));

Remember we added a new column address in the beginning? The above command will modify the address column of the **student** table, to now hold upto 300 characters.

ALTER Command: Rename a Column

Using ALTER command you can rename an existing column. Following is the syntax,

ALTER TABLE table_name RENAME old_column_name TO new_column_name;

Here is an example for this,

ALTER TABLE student RENAME address TO location;

The above command will rename address column to location.

ALTER Command: Drop a Column

ALTER command can also be used to drop or remove columns. Following is the syntax,

ALTER TABLE table_name DROP(column_name);

Here is an example for this,

ALTER TABLE student DROP(address);

The above command will drop the address column from the table **student**.

TRUNCATE command

TRUNCATE command removes all the records from a table. But this command will not destroy the table's structure. When we use TRUNCATE command on a table its (auto-increment) primary key is also initialized. Following is its syntax,

TRUNCATE TABLE table_name

Here is an example explaining it,

TRUNCATE TABLE student;

The above query will delete all the records from the table **student**.

In DML commands, we will study about the DELETE command which is also more or less same as the TRUNCATE command. We will also learn about the difference between the two in that tutorial.

DROP command

DROP command completely removes a table from the database. This command will also destroy the table structure and the data stored in it. Following is its syntax,

DROP TABLE table_name

Here is an example explaining it,

DROP TABLE student;

The above query will delete the **Student** table completely. It can also be used on Databases, to delete the complete database. For example, to drop a database,

DROP DATABASE Test;

The above query will drop the database with name **Test** from the system.

RENAME query

RENAME command is used to set a new name for any existing table. Following is the syntax,

RENAME TABLE old_table_name to new_table_name

Here is an example explaining it.

RENAME TABLE student to students_info;

The above query will rename the table **student** to **students_info**.

Use Database

Syntax : Use databasename;

DML(Data Manipulation Language)

INSERT SQL command

Data Manipulation Language (DML) statements are used for managing data in database. DML commands are not auto-committed. It means changes made by DML command are not permanent to database, it can be rolled back.

Talking about the Insert command, whenever we post a Tweet on Twitter, the text is stored in some table, and as we post a new tweet, a new record gets inserted in that table.

INSERT command

Insert command is used to insert data into a table. Following is its general syntax,

INSERT INTO table_name VALUES(data1, data2, ...)

Let's see an example,

Consider a table **student** with the following fields.

s_id	name	age

INSERT INTO student VALUES(101, 'Adam', 15);

The above command will insert a new record into **student** table.

s_id	name	age
101	Adam	15

Insert value into only specific columns

We can use the INSERT command to insert values for only some specific columns of a row. We can specify the column names along with the values to be inserted like this,

INSERT INTO student(id, name) values(102, 'Alex');

The above SQL query will only insert id and name values in the newly inserted record.

Insert NULL value to a column

Both the statements below will insert NULL value into **age** column of the **student** table.

INSERT INTO student(id, name) values(102, 'Alex');

Or,

INSERT INTO Student VALUES(102,'Alex', null);

The above command will insert only two column values and the other column is set to null.

S_id	S_Name	age
101	Adam	15
102	Alex	

Insert Default value to a column

INSERT INTO Student VALUES(103,'Chris', default)

S_id	S_Name	age
101	Adam	15
102	Alex	
103	chris	14

Suppose the column age in our tabel has a default value of 14.

Also, if you run the below query, it will insert default value into the age column, whatever the default value may be.

INSERT INTO Student VALUES(103,'Chris')

SELECT Command

SELECT query is used to retrieve data from a table. It is the most used SQL query. We can retrieve complete table data, or partial by specifying conditions using the WHERE clause.

Syntax of SELECT query

SELECT query is used to retrieve records from a table. We can specify the names of the columns which we want in the result set.

```
SELECT    column_name1,
          column_name2,
          column_name3,
          ...
          column_nameN
          FROM table_name;
```

Example

Consider the following **student** table,

s_id	name	age	address
101	Adam	15	Chennai
102	Alex	18	Delhi
103	Abhi	17	Banglore
104	Ankit	22	Mumbai

SELECT s_id, name, age FROM student;

The above query will fetch information of s_id, name and age columns of the **student** table and display them,

s_id	name	age
101	Adam	15
102	Alex	18
103	Abhi	17
104	Ankit	22

As you can see the data from address column is absent, because we did not specify it in our SELECT query.

Select all records from a table

A special character **asterisk** * is used to address all the data(belonging to all columns) in a query. SELECT statement uses * character to retrieve all records from a table, for all the columns.

SELECT * FROM student;

The above query will show all the records of **student** table, that means it will show complete dataset of the table.

s_id	name	age	address
101	Adam	15	Chennai
102	Alex	18	Delhi
103	Abhi	17	Banglore
104	Ankit	22	Mumbai

Select a particular record based on a condition

We can use the WHERE clause to set a condition,

```
SELECT * FROM student WHERE name = 'Abhi';
```

The above query will return the following result,

103	Abhi	17	Rohtak

Performing Simple Calculations using SELECT Query

Consider the following **employee** table.

eid	name	age	salary
101	Adam	26	5000
102	Ricky	42	8000
103	Abhi	25	10000
104	Rohan	22	5000

Here is our SELECT query,

SELECT eid, name, salary+3000 FROM employee;

The above command will display a new column in the result, with **3000** added into existing salaries of the employees.

eid	name	salary+3000
101	Adam	8000
102	Ricky	11000
103	Abhi	13000
104	Rohan	8000

So you can also perform simple mathematical operations on the data too using the SELECT query to fetch data from table.

UPDATE Command

Let's take an example of a real-world problem. These days, Facebook provides an option for **Editing** your status update, how do you think it works? Yes, using the **Update** SQL command.

Let's learn about the syntax and usage of the UPDATE command.

UPDATE command

UPDATE command is used to update any record of data in a table. Following is its general syntax,

UPDATE table_name SET column_name = new_value WHERE some_condition;

WHERE is used to add a condition to any SQL query, we will soon study about it in detail.

Let's take a sample table **student**,

student_id	name	age
101	Adam	15
102	Alex	
103	chris	14

UPDATE student SET age=18 WHERE student_id=102;

S_id	S_Name	age
101	Adam	15
102	Alex	18
103	chris	14

In the above statement, if we do not use the WHERE clause, then our update query will update age for all the columns of the table to **18**.

Updating Multiple Columns

We can also update values of multiple columns using a single UPDATE statement.

UPDATE student SET name='Abhi', age=17 where s_id=103;

The above command will update two columns of the record which has s_id 103.

s_id	name	age
101	Adam	15
102	Alex	18
103	Abhi	17

UPDATE Command: Incrementing Integer Value

When we have to update any integer value in a table, then we can fetch and update the value in the table in a single statement.

For example, if we have to update the age column of **student** table every year for every student, then we can simply run the following UPDATE statement to perform the following operation:

UPDATE student SET age = age+1;

As you can see, we have used age = age + 1 to increment the value of age by 1.

DELETE command

DELETE command is used to delete data from a table.

Following is its general syntax,

DELETE FROM table_name;

Let's take a sample table **student**:

s_id	name	age
101	Adam	15
102	Alex	18
103	Abhi	17

Delete all Records from a Table

DELETE FROM student;

The above command will delete all the records from the table **student**.

Delete a particular Record from a Table

In our **student** table if we want to delete a single record, we can use the WHERE clause to provide a condition in our DELETE statement.

DELETE FROM student WHERE s_id=103;

The above command will delete the record where s_id is 103 from the table **student**.

S_id	S_Name	age
101	Adam	15
102	Alex	18

Isn't DELETE same as TRUNCATE

TRUNCATE command is different from DELETE command. The delete command will delete all the rows from a table whereas truncate command not only deletes all the records stored in the table, but it also re-initializes the table(like a newly created table).

For eg: If you have a table with 10 rows and an **auto_increment** primary key, and if you use DELETEcommand to delete all the rows, it will delete all the rows, but will not re-initialize the primary key, hence if you will insert any row after using the DELETE command, the auto_increment primary key will start from 11. But in case of TRUNCATE command, primary key is re-initialized, and it will again start from 1.

WHERE 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.
When we specify a condition using the WHERE clause then the query executes only for those records for which the condition specified by the WHERE clause is true.

Syntax for WHERE clause

Here is how you can use the WHERE clause with a DELETE statement, or any other statement,
DELETE FROM table_name WHERE [condition];
The WHERE clause is used at the end of any SQL query, to specify a condition for execution.

Example

Consider a table **student**,

s_id	name	age	address
101	Adam	15	Chennai
102	Alex	18	Delhi
103	Abhi	17	Banglore
104	Ankit	22	Mumbai

Now we will use the SELECT statement to display data of the table, based on a condition, which we will add to our SELECT query using WHERE clause.
Let's write a simple SQL query to display the record for student with s_id as 101.

```
SELECT s_id,
   name,
   age,
   address
   FROM student WHERE s_id = 101;
```

Following will be the result of the above query.

s_id	name	age	address
101	Adam	15	Noida

Applying condition on Text Fields

In the above example we have applied a condition to an integer value field, but what if we want to apply the condition on name field. In that case we must enclose the value in single quote ' '. Some databases even accept double quotes, but single quotes is accepted by all.

```
SELECT s_id,
   name,
   age,
   address
   FROM student WHERE name = 'Adam';
```

Following will be the result of the above query.

s_id	name	age	address
101	Adam	15	Noida

SQL Data Types

SQL Data Type is an attribute that specifies the type of data of any object. Each column, variable and expression has a related data type in SQL.

Exact Numeric Data Types

DATA TYPE	FROM	TO
bigint	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
int	-2,147,483,648	2,147,483,647
smallint	-32,768	32,767
tinyint	0	255
bit	0	1
decimal	-10^38 +1	10^38 -1
numeric	-10^38 +1	10^38 -1
money	-922,337,203,685,477.5808	+922,337,203,685,477.5807
smallmoney	-214,748.3648	+214,748.3647

Approximate Numeric Data Types

DATA TYPE	FROM	TO
float	-1.79E + 308	1.79E + 308
real	-3.40E + 38	3.40E + 38

Date and Time Data Types

DATA TYPE	FROM	TO
datetime	Jan 1, 1753	Dec 31, 9999
smalldatetime	Jan 1, 1900	Jun 6, 2079
date	Stores a date like June 30, 1991	
time	Stores a time of day like 12:30 P.M.	

[**Note** − Here, datetime has 3.33 milliseconds accuracy where as smalldatetime has 1 minute accuracy.]

Character Strings Data Types

Sr.No.	DATA TYPE & Description
1	**char** Maximum length of 8,000 characters.(Fixed length non-Unicode characters)
2	**varchar** Maximum of 8,000 characters.(Variable-length non-Unicode data).
3	**varchar(max)** Maximum length of 2E + 31 characters, Variable-length non-Unicode data (SQL Server 2005 only).
4	**text** Variable-length non-Unicode data with a maximum length of 2,147,483,647 characters.

Unicode Character Strings Data Types

Sr.No.	DATA TYPE & Description
1	**nchar** Maximum length of 4,000 characters.(Fixed length Unicode)
2	**nvarchar** Maximum length of 4,000 characters.(Variable length Unicode)
3	**nvarchar(max)** Maximum length of 2E + 31 characters (SQL Server 2005 only).(Variable length Unicode)
4	**ntext** Maximum length of 1,073,741,823 characters. (Variable length Unicode)

Binary Data Types

Sr.No.	DATA TYPE & Description
1	**binary** Maximum length of 8,000 bytes(Fixed-length binary data)
2	**varbinary** Maximum length of 8,000 bytes.(Variable length binary data)
3	**varbinary(max)** Maximum length of 2E + 31 bytes (SQL Server 2005 only). (Variable length Binary data)
4	**image** Maximum length of 2,147,483,647 bytes. (Variable length Binary Data)

Misc Data Types

Sr.No.	DATA TYPE & Description
1	**sql_variant** Stores values of various SQL Server-supported data types, except text, ntext, and timestamp.
2	**timestamp** Stores a database-wide unique number that gets updated every time a row gets updated
3	**uniqueidentifier** Stores a globally unique identifier (GUID)
4	**xml** Stores XML data. You can store xml instances in a column or a variable (SQL Server 2005 only).
5	**cursor** Reference to a cursor object
6	**table** Stores a result set for later processing

Operator

An operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations. These Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

- Arithmetic operators
- Comparison operators
- Logical operators
- Operators used to negate conditions

SQL Arithmetic Operators

Assume **'variable a'** holds 10 and **'variable b'** holds 20, then −

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator.	a + b will give 30
- (Subtraction)	Subtracts right hand operand from left hand operand.	a - b will give -10
* (Multiplication)	Multiplies values on either side of the operator.	a * b will give 200
/ (Division)	Divides left hand operand by right hand operand.	b / a will give 2
% (Modulus)	Divides left hand operand by right hand operand and returns remainder.	b % a will give 0

Example 1

SQL> select 10+ 20;

Output

10 + 20
30

1 row in set (0.00 sec)

Example 2

SQL> select 10 * 20;

Output

10 * 20
200

1 row in set (0.00 sec)

Example 3

SQL> select 10 / 5;

Output

10 / 5
2.0000

1 row in set (0.03 sec)

Example 4

SQL> select 12 % 5;

Output

12 % 5
2

1 row in set (0.00 sec)

SQL Comparison Operators

Assume '**variable a**' holds 10 and '**variable b**' holds 20, then −

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(a = b) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.
!<	Checks if the value of left operand is not less than the value of right operand, if yes	(a !< b) is false.

	then condition becomes true.	
!>	Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true.	(a !> b) is true.

Consider the CUSTOMERS table having the following records −

SQL> SELECT * FROM CUSTOMERS;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khilan	25	Kupondole	1500.00
3	Kaushik	23	Kalimati	2000.00
4	Chaitali	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	10000.00

7 rows in set (0.00 sec)

Here are some simple examples showing the usage of SQL Comparison Operators −

Example 1

SQL> SELECT * FROM CUSTOMERS WHERE SALARY > 5000;

Output :

ID	NAME	AGE	ADDRESS	SALARY
4	Chaitali	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
7	Muffy	24	Gaushala	10000.00

3 rows in set (0.00 sec)

Example 2

SQL> SELECT * FROM CUSTOMERS WHERE SALARY = 2000;

Output

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
3	Kaushik	23	Kalimati	2000.00

2 rows in set (0.00 sec)

Example 3

SQL> SELECT * FROM CUSTOMERS WHERE SALARY != 2000;

Output

ID	NAME	AGE	ADDRESS	SALARY
2	Khilan	25	Kupondole	1500.00
4	Chaitali	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	10000.00

5 rows in set (0.00 sec)

Example 4

SQL> SELECT * FROM CUSTOMERS WHERE SALARY <> 2000;

Output

ID	NAME	AGE	ADDRESS	SALARY
2	Khilan	25	Kupondole	1500.00
4	Chaitali	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	10000.00

5 rows in set (0.00 sec)

Example 5

SQL> SELECT * FROM CUSTOMERS WHERE SALARY >= 6500;

Output

ID	NAME	AGE	ADDRESS	SALARY
4	Chaitali	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
7	Muffy	24	Gaushala	10000.00

3 rows in set (0.00 sec)

SQL Logical Operators

Here is a list of all the logical operators available in SQL.

Sr.No.	Operator & Description
1	**ALL** The ALL operator is used to compare a value to all values in another value set.
2	**AND** The AND operator allows the existence of multiple

	conditions in an SQL statement's WHERE clause.
3	**ANY** The ANY operator is used to compare a value to any applicable value in the list as per the condition.
4	**BETWEEN** The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.
5	**EXISTS** The EXISTS operator is used to search for the presence of a row in a specified table that meets a certain criterion.
6	**IN** The IN operator is used to compare a value to a list of literal values that have been specified.
7	**LIKE** The LIKE operator is used to compare a value to similar values using wildcard operators.
8	**NOT** The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.**
9	**OR** The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.
10	**IS NULL** The NULL operator is used to compare a value with a NULL value.
11	**UNIQUE** The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates).

Consider the CUSTOMERS table having the following records −

SQL> SELECT * FROM CUSTOMERS;

Output :

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khilan	25	Kupondole	1500.00
3	Kaushik	23	Kalimati	2000.00
4	Chaitali	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	10000.00

7 rows in set (0.00 sec)

Here are some simple examples showing usage of SQL Comparison Operators −

Example 1

SQL> SELECT * FROM CUSTOMERS WHERE AGE >= 25 AND SALARY >= 6500;

Output

ID	NAME	AGE	ADDRESS	SALARY
4	Chaitali	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00

2 rows in set (0.00 sec)

Example 2

SQL> SELECT * FROM CUSTOMERS WHERE AGE >= 25 OR SALARY >= 6500;

Output

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khilan	25	Kupondole	1500.00
4	Chaitali	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
7	Muffy	24	Gaushala	10000.00

5 rows in set (0.00 sec)

Example 3

SQL> SELECT * FROM CUSTOMERS WHERE AGE IS NOT NULL;

Output

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khilan	25	Kupondole	1500.00
3	Kaushik	23	Kalimati	2000.00
4	Chaitali	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	10000.00

7 rows in set (0.00 sec)

Example 4

SQL> SELECT * FROM CUSTOMERS WHERE NAME LIKE 'Ko%';

Output

ID	NAME	AGE	ADDRESS	SALARY
6	Komal	22	Teku	4500.00

1 row in set (0.00 sec)

Example 5

SQL> SELECT * FROM CUSTOMERS WHERE AGE IN (25, 27);

Output

ID	NAME	AGE	ADDRESS	SALARY
2	Khilan	25	Kupondole	1500.00
4	Chaitali	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00

3 rows in set (0.00 sec)

Example 6

SQL> SELECT * FROM CUSTOMERS WHERE AGE BETWEEN 25 AND 27;

Output

ID	NAME	AGE	ADDRESS	SALARY
2	Khilan	25	Kupondole	1500.00
4	Chaitali	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00

3 rows in set (0.00 sec)

Example 7

SQL> SELECT AGE FROM CUSTOMERS
WHERE EXISTS (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);

Output

AGE
32
25
23
25
27
22
24

7 rows in set (0.02 sec)

Example 8

SQL> SELECT * FROM CUSTOMERS

WHERE AGE > ALL (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);

Output

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00

1 row in set (0.02 sec)

Example 9

SQL> SELECT * FROM CUSTOMERS

WHERE AGE > ANY (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);

Output

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khilan	25	Kupondole	1500.00
4	Chaitali	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00

4 rows in set (0.00 sec)

SQL CONSTRAINTS

SQL Constraints are rules used to limit the type of data that can go into a table, to maintain the accuracy and integrity of the data inside table.

Constraints can be divided into the following two types,

- **Column level constraints:** Limits only column data.
- **Table level constraints:** Limits whole table data.

Constraints are used to make sure that the integrity of data is maintained in the database. Following are the most used constraints that can be applied to a table.

- NOT NULL
- UNIQUE
- PRIMARY KEY
- FOREIGN KEY
- CHECK
- DEFAULT

NOT NULL Constraint

NOT NULL constraint restricts a column from having a NULL value. Once **NOT NULL** constraint is applied to a column, you cannot pass a null value to that column. It enforces a column to contain a proper value.

One important point to note about this constraint is that it cannot be defined at table level.

Example using NOT NULL constraint

```
CREATE TABLE CUSTOMERS(
   ID   INT  NOT NULL,
   NAME VARCHAR (20)  NOT NULL,
   AGE  INT  NOT NULL,
   ADDRESS  CHAR (25) ,
   SALARY  DECIMAL (18, 2),
   PRIMARY KEY (ID)
);
```

If CUSTOMERS table has already been created, then to add a NOT NULL constraint to the SALARY column in Oracle and MySQL, you would write a query like the one that is shown in the following code block.

```
ALTER TABLE CUSTOMERS
   MODIFY SALARY  DECIMAL (18, 2) NOT NULL;
```

UNIQUE Constraint

UNIQUE constraint ensures that a field or column will only have unique values. A **UNIQUE** constraint field will not have duplicate data. This constraint can be applied at column level or table level.

```
CREATE TABLE CUSTOMERS(
   ID   INT          NOT NULL,
   NAME VARCHAR (20)    NOT NULL,
   AGE  INT          NOT NULL UNIQUE,
   ADDRESS  CHAR (25) ,
   SALARY   DECIMAL (18, 2),
   PRIMARY KEY (ID)
);
```

If the CUSTOMERS table has already been created, then to add a UNIQUE constraint to the AGE column. You would write a statement like the query that is given in the code block below.

```
ALTER TABLE CUSTOMERS
MODIFY AGE INT NOT NULL UNIQUE;
```

You can also use the following syntax, which supports naming the constraint in multiple columns as well.

```
ALTER TABLE CUSTOMERS
   ADD CONSTRAINT myUniqueConstraint UNIQUE(AGE, SALARY);
```

DROP a UNIQUE Constraint

To drop a UNIQUE constraint, use the following SQL query.

```
ALTER TABLE CUSTOMERS
   DROP CONSTRAINT myUniqueConstraint;
```

If you are using MySQL, then you can use the following syntax −

```
ALTER TABLE CUSTOMERS
  DROP INDEX myUniqueConstraint;
```

Primary Key Constraint

Primary key constraint uniquely identifies each record in a database. A Primary Key must contain unique value and it must not contain null value. Usually Primary Key is used to index the data inside the table.

Create Primary Key

Here is the syntax to define the ID attribute as a primary key in a CUSTOMERS table.

```
CREATE TABLE CUSTOMERS(
        ID   INT              NOT NULL,
        NAME VARCHAR (20)     NOT NULL,
        AGE  INT              NOT NULL,
        ADDRESS  CHAR (25) ,
        SALARY   DECIMAL (18, 2),
        PRIMARY KEY (ID)
        );
```

To create a PRIMARY KEY constraint on the "ID" column when the CUSTOMERS table already exists, use the following SQL syntax −

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

NOTE − If you use the ALTER TABLE statement to add a primary key, the primary key column(s) should have already been declared to not contain NULL values (when the table was first created).

For defining a PRIMARY KEY constraint on multiple columns, use the SQL syntax given below.

```
CREATE TABLE CUSTOMERS(
        ID   INT              NOT NULL,
        NAME VARCHAR (20)     NOT NULL,
        AGE  INT              NOT NULL,
        ADDRESS  CHAR (25) ,
        SALARY   DECIMAL (18, 2),
        PRIMARY KEY (ID, NAME)
        );
```

To create a PRIMARY KEY constraint on the "ID" and "NAMES" columns when CUSTOMERS table already exists, use the following SQL syntax.

```
ALTER TABLE CUSTOMERS
        ADD CONSTRAINT PK_CUSTID PRIMARY KEY (ID, NAME);
```

Delete Primary Key

You can clear the primary key constraints from the table with the syntax given below.

```
ALTER TABLE CUSTOMERS DROP PRIMARY KEY ;
```

Foreign Key Constraint

FOREIGN KEY is used to relate two tables. FOREIGN KEY constraint is also used to restrict actions that would destroy links between tables. To understand FOREIGN KEY, let's see its use, with help of the below tables:

Customer_Detail Table

c_id	Customer_Name	address
101	Adam	Noida
102	Alex	Delhi
103	Stuart	Rohtak

Order_Detail Table

Order_id	Order_Name	c_id
10	Order1	101
11	Order2	103
12	Order3	102

In **Customer_Detail** table, **c_id** is the primary key which is set as foreign key in **Order_Detail** table. The value that is entered in **c_id** which is set as foreign key in **Order_Detail** table must be present in **Customer_Detail** table where it is set as primary key. This prevents invalid data to be inserted into **c_id** column of **Order_Detail** table.

If you try to insert any incorrect data, DBMS will return error and will not allow you to insert the data.

Using FOREIGN KEY constraint at Table Level

```
CREATE table Order_Detail(
   order_id int PRIMARY KEY,
   order_name varchar(60) NOT NULL,
   c_id int FOREIGN KEY REFERENCES Customer_Detail(c_id)
);
```

In this query, **c_id** in table Order_Detail is made as foriegn key, which is a reference of **c_id** column in Customer_Detail table.

Using FOREIGN KEY constraint at Column Level

```
ALTER table Order_Detail ADD FOREIGN KEY (c_id) REFERENCES Customer_Detail(c_id);
```

Behaviour of Foriegn Key Column on Delete

There are two ways to maintin the integrity of data in Child table, when a particular record is deleted in the main table. When two tables are connected with Foriegn key, and certain data in the main table is deleted, for which a record exits in the child table, then we must have some mechanism to save the integrity of data in the child table.

1. **On Delete Cascade :** This will remove the record from child table, if that value of foriegn key is deleted from the main table.
2. **On Delete Null :** This will set all the values in that record of child table as NULL, for which the value of foriegn key is deleted from the main table.
3. If we don't use any of the above, then we cannot delete data from the main table for which data in child table exists. We will get an error if we try to do so.

ERROR : record in child table exist

CHECK Constraint

CHECK constraint is used to restrict the value of a column between a range. It performs check on the values, before storing them into the database. Its like condition checking before saving data into a column.

Example

For example, the following program creates a new table called CUSTOMERS and adds five columns. Here, we add a CHECK with AGE column, so that you cannot have any CUSTOMER who is below 18 years.

```
CREATE TABLE CUSTOMERS(
        ID   INT          NOT NULL,
        NAME VARCHAR (20)    NOT NULL,
        AGE  INT          NOT NULL CHECK (AGE >= 18),
        ADDRESS  CHAR (25) ,
        SALARY   DECIMAL (18, 2),
        PRIMARY KEY (ID)
        );
```

If the CUSTOMERS table has already been created, then to add a CHECK constraint to AGE column, you would write a statement like the one given below.

```
ALTER TABLE CUSTOMERS
        MODIFY AGE INT NOT NULL CHECK (AGE >= 18 );
```

You can also use the following syntax, which supports naming the constraint in multiple columns as well −

```
ALTER TABLE CUSTOMERS
        ADD CONSTRAINT myCheckConstraint CHECK(AGE >= 18);
```

DROP a CHECK Constraint

To drop a CHECK constraint, use the following SQL syntax. This syntax does not work with MySQL.

```
ALTER TABLE CUSTOMERS
        DROP CONSTRAINT myCheckConstraint;
```

DEFAULT Constraints

The DEFAULT constraint provides a default value to a column when the INSERT INTO statement does not provide a specific value.

Example

For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, the SALARY column is set to 5000.00 by default, so in case the INSERT INTO statement does not provide a value for this column, then by default this column would be set to 5000.00.

```
CREATE TABLE CUSTOMERS(
        ID   INT            NOT NULL
        NAME VARCHAR (20)     NOT NULL,
        AGE  INT            NOT NULL,
        ADDRESS  CHAR (25) ,
        SALARY   DECIMAL (18, 2) DEFAULT 5000.00,
        PRIMARY KEY (ID)
);
```

If the CUSTOMERS table has already been created, then to add a DEFAULT constraint to the SALARY column, you would write a query like the one which is shown in the code block below.

```
ALTER TABLE CUSTOMERS
        MODIFY SALARY  DECIMAL (18, 2) DEFAULT 5000.00;
```

Drop Default Constraint

To drop a DEFAULT constraint, use the following SQL query.

```
ALTER TABLE CUSTOMERS
        ALTER COLUMN SALARY DROP DEFAULT;
```

INDEX Constraints

The INDEX is used to create and retrieve data from the database very quickly. An Index can be created by using a single or group of columns in a table. When the index is created, it is assigned a ROWID for each row before it sorts out the data.

Proper indexes are good for performance in large databases, but you need to be careful while creating an index. A Selection of fields depends on what you are using in your SQL queries.

Example

For example, the following SQL syntax creates a new table called CUSTOMERS and adds five columns in it.

```
CREATE TABLE CUSTOMERS(ID   INT            NOT NULL,
        NAME VARCHAR (20)     NOT NULL, AGE  INT            NOT
        NULL, ADDRESS  CHAR (25) ,
        SALARY   DECIMAL (18, 2),
        PRIMARY KEY (ID));
```

Now, you can create an index on a single or multiple columns using the syntax given below.

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

To create an INDEX on the AGE column, to optimize the search on customers for a specific age, you can use the follow SQL syntax which is given below −

```
CREATE INDEX idx_age
        ON CUSTOMERS ( AGE );
```

DROP an INDEX Constraint

To drop an INDEX constraint, use the following SQL syntax.

```
ALTER TABLE CUSTOMERS
        DROP INDEX idx_age;
```

Order By Clause

It is used to sort data in ascending or descending order based on one or more columns.

Syntax:
```
Select column_list
From tablename
[Where condition]
[order by column1, column2, .... , columnN][ASC|DESC];
```

Example :
```
Select * from customer
Order by Name, Salary;

Select * from customer
Order by Name DESC;
```

Group By Clause

It is used in collaboration with the SELECT statement to arrange identical data into groups. It follows WHERE clause in SELECT statements and precedes the order by clause.

Syntax :
```
Select column1, column2
From table-name
[Where condition]
[Group by column1, column2]
[Order by column1, column2]
```

Example :
```
Select Name, Sum(Salary)
From customer
Group By Name
```

Distinct Keyword

It used in conjunction with the SELECT statements to eliminate all the duplicate records and fetching only unique records. Multiple duplicate records in table will be eliminated.

Syntax :
```
Select DISTINCT column1, ...., columnN
From table-name
[Where condition]
```

Example : Select Distinct Salary From customer Order By Salary;

Having Clause

It filters records that work on summarized GROUP BY results. It applies to summarized group records, whereas WHERE applies to individual records. Only the groups that meet the HAVING criteria will be returned. Having requires that a GROUP BY clause is present. WHERE and HAVING can be in the same query.

Syntax :
```
Select column-names
From table-name
Where condition
Group By column-names
Having condition
```
With Order By clause
```
Select column-names
From table-name
Where condition
Group By column-names
Having condition
Order  By column-names
```
Example :
```
Select COUNT(ID), Country
From customer
Group By Country
Having COUNT(ID) > 10
```
Top Clause

It is used to fetch a TOP N number or X percent records from the table.

[Note : MySql does not support TOP clause, MySQL supports LIMIT clause and Oracle supports Rownum command]

Syntax :
```
Select Top number/percent column-namse
From table-name
[Where condition];
```
Example :
```
Select TOP 3 * from Customer; [For sql server]
Select * from customer limit 3; [For MySQL]
Select *from customer Where Rownum <=3; [For oracle]
```

Sub-Queries

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause.

A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

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.

Subqueries with the SELECT Statement

Subqueries are most frequently used with the SELECT statement. The basic syntax is as follows

```
SELECT column_name [, column_name ]
FROM   table1 [, table2 ]
WHERE  column_name OPERATOR
  (SELECT column_name [, column_name ]
  FROM table1 [, table2 ]
  [WHERE])
```

Example

Consider the CUSTOMERS table having the following records −

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khilan	25	Kupondole	1500.00
3	Kaushik	23	Kalimati	2000.00
4	Chaitali	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	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	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
7	Muffy	24	Gaushala	10000.00

Subqueries with the INSERT Statement

Subqueries also can be used with INSERT statements. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery can be modified with any of the character, date or number functions.

The basic syntax is as follows.

```
INSERT INTO table_name [ (column1 [, column2 ]) ]
  SELECT [ *|column1 [, column2 ]
  FROM table1 [, table2 ]
  [ WHERE VALUE OPERATOR ]
```

Example

Consider a table CUSTOMERS_BKP with similar structure as CUSTOMERS table. Now to copy the complete CUSTOMERS table into the CUSTOMERS_BKP table, you can use the following syntax.

```
SQL> INSERT INTO CUSTOMERS_BKP
  SELECT * FROM CUSTOMERS
  WHERE ID IN (SELECT ID
  FROM CUSTOMERS) ;
```

Subqueries with the UPDATE Statement

The subquery can be used in conjunction with the UPDATE statement. Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement.

The basic syntax is as follows.

```
UPDATE table
SET column_name = new_value
[ WHERE OPERATOR [ VALUE ]
  (SELECT COLUMN_NAME
```

FROM TABLE_NAME)

[WHERE)]

Example

Assuming, we have CUSTOMERS_BKP table available which is backup of CUSTOMERS table. The following example updates SALARY by 0.25 times in the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

```
SQL> UPDATE CUSTOMERS
  SET SALARY = SALARY * 0.25
  WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP
    WHERE AGE >= 27 );
```

This would impact two rows and finally CUSTOMERS table would have the following records.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khilan	25	Kupondole	1500.00
3	Kaushik	23	Kalimati	2000.00
4	Chaitali	25	Kalanki	6500.00
5	Hardik	27	Tripureshwor	8500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	10000.00

Subqueries with the DELETE Statement

The subquery can be used in conjunction with the DELETE statement like with any other statements mentioned above.

The basic syntax is as follows.

```
DELETE FROM TABLE_NAME
[ WHERE OPERATOR [ VALUE ]
  (SELECT COLUMN_NAME
  FROM TABLE_NAME)
  [ WHERE) ]
```

Example

Assuming, we have a CUSTOMERS_BKP table available which is a backup of the CUSTOMERS table. The following example deletes the records from the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

```
SQL> DELETE FROM CUSTOMERS
  WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP
    WHERE AGE >= 27 );
```

This would impact two rows and finally the CUSTOMERS table would have the following records.

ID	NAME	AGE	ADDRESS	SALARY
2	Khilan	25	Kupondole	1500.00
3	Kaushik	23	Kalimati	2000.00
4	Chaitali	25	Kalanki	6500.00
6	Komal	22	Teku	4500.00
7	Muffy	24	Gaushala	10000.00

SQL Functions

SQL provides many built-in functions to perform operations on data. These functions are useful while performing mathematical calculations, string concatenations, sub-strings etc. SQL functions are divided into two categories,

1. Aggregate Functions
2. Scalar Functions

Aggregate Function

It is used to accumulate information from multiple tuples forming a single tuple summary.

AVG()Function

Average returns average value after calculating it from values in a numeric column.

Its general **syntax** is,

SELECT AVG(column_name) FROM table_name

Using AVG() function

Consider the following **Emp** table

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	10000
405	Tiger	35	8000

SQL query to find average salary will be,

SELECT avg(salary) from Emp;

Result of the above query will be,

avg(salary)
8200

COUNT() Function

Count returns the number of rows present in the table either based on some condition or without condition.

Its general **syntax** is,

SELECT COUNT(column_name) FROM table-name

Using COUNT() function

Consider the following **Emp** table

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	10000
405	Tiger	35	8000

SQL query to count employees, satisfying specified condition is,

SELECT COUNT(name) FROM Emp WHERE salary = 8000;

Result of the above query will be,

count(name)
2

Example of COUNT(distinct)

Consider the following **Emp** table

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	10000
405	Tiger	35	8000

SQL query is,

SELECT COUNT(DISTINCT salary) FROM emp;

Result of the above query will be,

count(distinct salary)
4

FIRST() Function

First function returns first value of a selected column

Syntax for FIRST function is,

SELECT FIRST(column_name) FROM table-name;

Using FIRST() function

Consider the following **Emp** table

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	10000
405	Tiger	35	8000

SQL query will be,
SELECT FIRST(salary) FROM Emp;
and the result will be,

first(salary)
9000

LAST() Function

LAST function returns the return last value of the selected column.

Syntax of LAST function is,

SELECT LAST(column_name) FROM table-name;

Using LAST() function

Consider the following **Emp** table

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	10000
405	Tiger	35	8000

SQL query will be,
SELECT LAST(salary) FROM emp;
Result of the above query will be,

last(salary)
8000

MAX() Function

MAX function returns maximum value from selected column of the table.

Syntax of MAX function is,

SELECT MAX(column_name) from table-name;

Using MAX() function

Consider the following **Emp** table

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	10000
405	Tiger	35	8000

SQL query to find the Maximum salary will be,
SELECT MAX(salary) FROM emp;
Result of the above query will be,

MAX(salary)
10000

MIN() Function

MIN function returns minimum value from a selected column of the table.

Syntax for MIN function is,

SELECT MIN(column_name) from table-name;

Using MIN() function

Consider the following **Emp** table,

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	10000
405	Tiger	35	8000

SQL query to find minimum salary is,

SELECT MIN(salary) FROM emp;

Result will be,

MIN(salary)
6000

SUM() Function

SUM function returns total sum of a selected columns numeric values.

Syntax for SUM is,

SELECT SUM(column_name) from table-name;

Using SUM() function

Consider the following **Emp** table

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	10000
405	Tiger	35	8000

SQL query to find sum of salaries will be,

SELECT SUM(salary) FROM emp;

Result of above query is,

SUM(salary)
41000

Like Clause

The SQL Server LIKE is a logical operator that determines if a character string matches a specified pattern. A pattern may include regular characters and wildcard characters. The LIKE operator is used in the WHERE clause of the SELECT, UPDATE and DELETE statements to filter rows based on pattern matching.

The following illustrates the syntax of the SQL Server LIKE operator:

1 column | expression LIKE patterns [ESCAPE escape_chapter]

Pattern

The pattern is a sequence of characters to search for in the column or expression. It can include the following valid wildcard characters:

- The percent wildcard (%): any string of zero or more characters.
- The underscore (_) wildcard: any single character.
- The [list of characters] wildcard: any single character within the specified set.
- The [character-character]: any single character within the specified range.
- The [^]: any single character not within a list or a range.

The wildcard characters makes the LIKE operator more flexible than the equal (=) and not equal (!=) string comparison operators.

Escape character

The escape character instructs the LIKE operator to treat the wildcard characters as the regular characters. The escape character has no default value and must be evaluated to only one character.

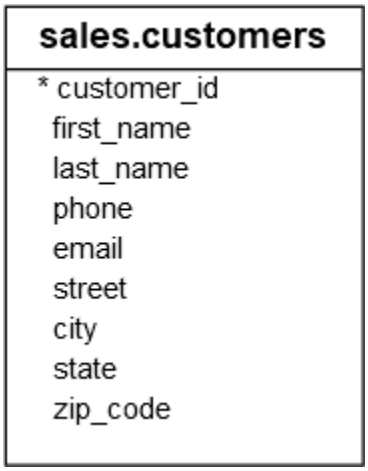
The LIKE operator returns TRUE if the column or expression matches the specified pattern.

To negate the result of the LIKE operator, you use the NOT operator as follows:

1 column | expression NOT LIKE pattern [ESCAPE escape_character]

SQL Server LIKE examples

See the following customers table from the sample database:



The % (percent) wildcard examples

The following example finds the customers whose last name starts with the letter z:

```
Select customer_id, first_name, last_name FROM sales.customers
Where last_name LIKE 'z%'
Order By first_name;
```

customer_id	first_name	last_name
1354	Alexandria	Zamora
304	Jayme	Zamora
110	Ollie	Zimmerman

The following example returns the customers whose last name ends with the string er:

```
SELECT customer_id, first_name, last_name FROM sales.customers
WHERE last_name LIKE '%er'
ORDER BY first_name;
```

customer_id	first_name	last_name
1412	Adrien	Hunter
62	Alica	Hunter
619	Ana	Palmer
525	Andreas	Mayer
528	Angele	Schroeder
1345	Arie	Hunter
851	Arlena	Buckner
477	Arminda	Weber
425	Augustina	Joyner
290	Barry	Buckner
1169	Beatris	Joyner

The following statement retrieves the customers whose last name starts with the letter t and ends with the letter s:

```
SELECT    customer_id,    first_name,    last_name
FROM    sales.customers
WHERE   last_name LIKE 't%s'
ORDER BY    first_name;
```

customer_id	first_name	last_name
682	Amita	Thomas
904	Jana	Thomas
1360	Latashia	Travis
567	Sheila	Travis

The _ (underscore) wildcard example

The underscore represents a single character. For example, the following statement returns the customers where the second character is the letter u:

```
SELECT  customer_id,  first_name,  last_name
FROM  sales.customers
WHERE  last_name LIKE '_u%'
ORDER BY  first_name;
```

customer_id	first_name	last_name
338	Abbey	Pugh
1412	Adrien	Hunter
527	Afton	Juarez
442	Alane	Munoz
62	Alica	Hunter
683	Amparo	Burks
1350	Annett	Rush
1345	Arie	Hunter
851	Arlena	Buckner
1200	Aubrey	Durham
290	Barry	Buckner

The pattern_u%

- The first underscore character (_) matches any single character.
- The second letter u matches the letter u exactly
- The third character % matches any sequence of characters

The [list of characters] wildcard example

The square brackets with a list of characters e.g. [ABC] represent a single character that must be one of the characters specified in the list.

For example, the following query returns the customers where the first character in the last name is Y or Z:

```
SELECT  customer_id,  first_name,  last_name
FROM  sales.customers
WHERE  last_name LIKE '[YZ]%'
ORDER BY  last_name
```

customer_id	first_name	last_name
54	Fran	Yang
250	Ivonne	Yang
768	Yvone	Yates
223	Scarlet	Yates
498	Edda	Young
543	Jasmin	Young
1354	Alexandria	Zamora
304	Jayme	Zamora
110	Ollie	Zimmerman

The [character-character] wildcard example

The square brackets with a character range e.g., [A-C] represent a single character that must be within a specified range.

For example, the following query finds the customers where the first character in the last name is the letter in the range A through C:

```
SELECT customer_id, first_name, last_name
FROM sales.customers
WHERE last_name LIKE '[A-C]%'
ORDER BY first_name;
```

customer_id	first_name	last_name
1224	Abram	Copeland
1023	Adena	Blake
1061	Alanna	Barry
1219	Alden	Atkinson
1135	Alisia	Albert
892	Alissa	Craft
1288	Allie	Conley
1295	Alline	Beasley
1168	Almeta	Benjamin
683	Amparo	Burks
947	Angele	Castro

The [^Character List or Range] wildcard example

The square brackets with a caret sign (^) followed by a range e.g., [^A-C] or character list e.g., [ABC] represent a single character that is not in the specified range or character list.

For example, the following query returns the customers where the first character in the last name is not the letter in the range A through X:

```
SELECT customer_id, first_name, last_name
FROM sales.customers
WHERE last_name LIKE '[^A-X]%'
ORDER BY last_name;
```

customer_id	first_name	last_name
54	Fran	Yang
250	Ivonne	Yang
768	Yvone	Yates
223	Scarlet	Yates
498	Edda	Young
543	Jasmin	Young
1354	Alexandria	Zamora
304	Jayme	Zamora
110	Ollie	Zimmerman

The NOT LIKE operator example

The following example uses the NOT LIKE operator to find customers where the first character in the first name is not the letter A:

```
SELECT customer_id, first_name, last_name
FROM sales.customers
WHERE first_name NOT LIKE 'A%'
ORDER BY first_name;
```

customer_id	first_name	last_name
174	Babara	Ochoa
1108	Bao	Wade
225	Barbera	Riggs
1249	Barbra	Dickerson
802	Barrett	Sanders
1154	Barry	Albert
290	Barry	Buckner
399	Bart	Hess
269	Barton	Crosby
977	Barton	Cox

[Note : (_u%) - Name has second character
([YZ]%) - First character is Y or Z
([A-C]%) - First character range from A through C
(%K) - Name ends with k
(t%s) - Name starts with 't' and end with 's'
(s%) - Name starts with 's'
(2_%_%) - starts with 2 and are at least 3 characters in length
(_2%3) - 2 in second position and end with 3
(NOT LIKE A%) - first character is not 'A'
(%A%) - A in any position]

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

UNION Operation

UNION is used to combine the results of two or more SELECT statements. However it will eliminate duplicate rows from its resultset. In case of union, number of columns and data type must be same in both the tables, on which UNION operation is being applied.

Example of UNION

The **First** table,

ID	Name
1	abhi
2	adam

The **Second** table,

ID	Name
2	adam
3	Chester

Union SQL query will be,

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

The result set table will look like,

ID	NAME
1	abhi
2	adam
3	Chester

UNION ALL

This operation is similar to Union. But it also shows the duplicate rows.

Example of Union All

The **First** table,

ID	NAME
1	abhi
2	adam

The **Second** table,

ID	NAME
2	adam
3	Chester

Union All query will be like,

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

The result set table will look like,

ID	NAME
1	abhi
2	adam
2	adam
3	Chester

INTERSECT

Intersect operation is used to combine two SELECT statements, but it only returns the records which are common from both SELECT statements. In case of **Intersect** the number of columns and data type must be same.

[**NOTE:** MySQL does not support INTERSECT operator.]

Example of Intersect

The **First** table,

ID	NAME
1	abhi
2	adam

The **Second** table,

ID	NAME
2	adam
3	Chester

Intersect query will be,

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

The resultset table will look like

ID	NAME
2	adam

MINUS

The Minus operation combines results of two SELECT statements and return only those in the final result, which belongs to the first set of the result.

Example of Minus

The **First** table,

ID	NAME
1	abhi
2	adam

The **Second** table,

ID	NAME
2	adam
3	Chester

Minus query will be,

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

The result set table will look like,

ID	NAME
1	abhi

Relations (Joined & Derived)

SQL join

A SQL join is a Structured Query Language (**SQL**) instruction to combine data from two sets of data (i.e. two tables). Before we dive into the details of a SQL join, let's briefly discuss what SQL is, and why someone would want to perform a SQL join.

SQL is a special-purpose programming language designed for managing information in a relational database management system (**RDBMS**). The word relational here is key; it specifies that the database management system is organized in such a way that there are clear relations defined between different sets of data.

Typically, you need to extract, transform, and load data into your RDBMS before you're able to manage it using SQL, which you can accomplish by using a tool like Stitch.

Example

Imagine you're running a store and would like to record information about your customers and their orders. By using a relational database, you can save this information as two tables that represent two distinct entities: customers and orders.

Customers

customer_id	first_name	last_name	email	address	city	state	zip
1	George	Washington	gwashington@usa.gov	3200 Mt Vernon Hwy	Mount Vernon	VA	22121
2	John	Adams	jadams@usa.gov	1250 Hancock St	Quincy	MA	02169
3	Thomas	Jefferson	tjefferson@usa.gov	931 Thomas Jefferson Pkwy	Charlottesville	VA	22902
4	James	Madison	jmadison@usa.gov	11350 Constitution Hwy	Orange	VA	22960
5	James	Monroe	jmonroe@usa.gov	2050 James Monroe Pkwy	Charlottesville	VA	22902

Here, information about each customer is stored in its own row, with columns specifying different bits of information, including their first name, last name, and email address. Additionally, we associate a unique customer number, or primary key, with each customer record.

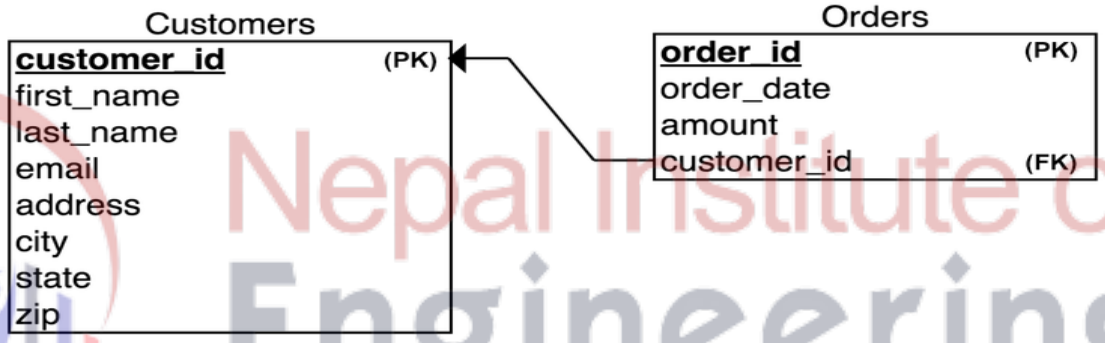
Orders

order_id	order_date	amount	customer_id
1	07/04/1776	$234.56	1
2	03/14/1760	$78.50	3
3	05/23/1784	$124.00	2
4	09/03/1790	$65.50	3
5	07/21/1795	$25.50	10
6	11/27/1787	$14.40	9

Again, each row contains information about a specific order. Each order has its own unique identification key – order_id for for this table – assigned to it as well.

Relational Model

You've probably noticed that these two examples share similar information. You can see these simple relations diagrammed below:



Note that the orders table contains two keys: one for the order and one for the customer who placed that order. In scenarios when there are multiple keys in a table, the key that refers to the entity being described in that table is called the **primary key** (PK) and other key is called a **foreign key** (FK).

In our example, order_id is a primary key in the orders table, while customer_id is both a primary key in the customers table and a foreign key in the orders table. Primary and foreign keys are essential to describing relations between the tables, and in performing SQL joins.

SQL Join Example

Let's say we want to find all orders placed by a particular customer. We can do this by joining the customers and orders tables together using the relationship established by the customer_id key :

```
Select order_date, order_amount
From customers
join orders
on customers.customer_id = orders.customer_id
AND customer_id = 3
```

Here, we're joining the two tables using the join keyword, and specifying what key to use when joining the tables in the on customer.customer_id = orders.customer_id line following the join

statement. Here is the result of the above SQL query, which includes two orders placed by Thomas Jefferson (customer_id = 3):

order_id	order_date	order_amount
2	3/14/1760	$78.50
4	9/03/1790	$65.50

This particular join is an example of an “inner” join. Depending on the kind of analysis you’d like to perform, you may want to use a different method. There are actually a number of different ways to join the two tables together, depending on your application. The next section will explain inner, left, right, and full joins, and provide examples using the data tables used above.

Basic SQL Join Types

There are four basic types of SQL joins: inner, left, right, and full. The easiest and most intuitive way to explain the difference between these four types is by using a Venn diagram, which shows all possible logical relations between data sets.

Again, it's important to stress that before you can begin using any join type, you'll need to extract the data and load it into an RDBMS like Amazon Redshift, where you can query tables from multiple sources. You build that process manually, or you can use an ETL service like Stitch,

Let’s say we have two sets of data in our relational database: table A and table B, with some sort of relation specified by primary and foreign keys. The result of joining these tables together can be visually represented by the following diagram:

The extent of the overlap, if any, is determined by how many records in Table A match the records in Table B. Depending on what subset of data we would like to select from the two tables, the four join types can be visualized by highlighting the corresponding sections of the Venn diagram:

Select all records from Table A and Table B, where the join condition is met.

Select all records from Table A, along with records from Table B for which the join condition is met (if at all).

Select all records from Table B, along with records from Table A for which the join condition is met (if at all).

Select all records from Table A and Table B, regardless of whether the join condition is met or not.

Cross JOIN or Cartesian Product

This type of JOIN returns the Cartesian product of rows from the tables in Join. It will return a table which consists of records which combines each row from the first table with each row of the second table.

Cross JOIN Syntax is,

```
SELECT column-name-list
FROM
table-name1 CROSS JOIN table-name2;
```

Example of Cross JOIN

Following is the class table,

ID	NAME
1	abhi
2	adam
4	alex

and the class_info table,

ID	Address
1	Kathmandu
2	Kupondole
3	Kalimati

CROSS JOIN query will be,

SELECT * FROM
class CROSS JOIN class_info;

The resultset table will look like,

ID	NAME	ID	Address
1	abhi	1	Kathmandu
2	adam	1	Kathmandu
4	alex	1	Kathmandu
1	abhi	2	Kupondole
2	adam	2	Kupondole
4	alex	2	Kupondole
1	abhi	3	Kalimati
2	adam	3	Kalimati
4	alex	3	Kalimati

As you can see, this join returns the cross product of all the records present in both the tables.

INNER Join or EQUI Join

This is a simple JOIN in which the result is based on matched data as per the equality condition specified in the SQL query.

Inner Join Syntax is,

SELECT column-name-list FROM
table-name1 INNER JOIN table-name2
ON table-name1.column-name = table-name2.column-name;

Example of INNER JOIN

Consider a **class** table,

ID	NAME
1	abhi
2	adam
4	alex

and the **class_info** table,

ID	Address
1	Kathmandu
2	Kupondole
3	Kalimati

Inner JOIN query will be,

SELECT * from class INNER JOIN class_info where class.id = class_info.id;

The resultset table will look like,

ID	NAME	ID	Address
1	abhi	1	Kathmandu
2	adam	2	Kupondole
3	alex	3	Kalimati

Natural JOIN

Natural Join is a type of Inner join which is based on column having same name and same datatype present in both the tables to be joined.

The syntax for Natural Join is,

SELECT * FROM
table-name1 NATURAL JOIN table-name2;

Example of Natural JOIN

Here is the **class** table,

Consider a **class** table,

ID	NAME
1	abhi
2	adam
4	alex

and the **class_info** table,

ID	Address
1	Kathmandu
2	Kupondole
3	Kalimati

Natural join query will be,

SELECT * from class NATURAL JOIN class_info;

The resultset table will look like,

ID	NAME	ID	Address
1	abhi	1	Kathmandu
2	adam	2	Kupondole
3	alex	3	Kalimati

In the above example, both the tables being joined have **ID** column(same name and same data type), hence the records for which value of **ID** matches in both the tables will be the result of Natural Join of these two tables.

OUTER JOIN

Outer Join is based on both matched and unmatched data. Outer Joins subdivide further into,

1. Left Outer Join
2. Right Outer Join
3. Full Outer Join

LEFT Outer Join

The left outer join returns a resultset table with the matched data from the two tables and then the remaining rows of the left table and null from the right table's columns.

Syntax for Left Outer Join is,

```
SELECT column-name-list FROM
table-name1 LEFT OUTER JOIN table-name2
ON table-name1.column-name = table-name2.column-name;
```

To specify a condition, we use the ON keyword with Outer Join.

Left outer Join Syntax for Oracle is,

```
SELECT column-name-list FROM
table-name1, table-name2 on table-name1.column-name = table-name2.column-name(+);
```

Example of Left Outer Join

Here is the **class** table,

ID	NAME
1	abhi
2	adam
3	alex
4	anu
5	ashish

and the **class_info** table,

ID	Address
1	Kathmandu
2	Bhaktpur
3	Kalimati
7	Teku
8	Tripureshwor

Left Outer Join query will be,

```
SELECT * FROM class LEFT OUTER JOIN class_info ON (class.id = class_info.id);
```

The resultset table will look like,

ID	NAME	ID	Address
1	abhi	1	Kathmandu
2	adam	2	Kupndole

3	alex	3	Kalimati
4	anu	null	null
5	ashish	null	null

RIGHT Outer Join

The right outer join returns a resultset table with the **matched data** from the two tables being joined, then the remaining rows of the **right** table and null for the remaining **left** table's columns.

Syntax for Right Outer Join is,

```
SELECT column-name-list FROM
table-name1 RIGHT OUTER JOIN table-name2
ON table-name1.column-name = table-name2.column-name;
```

Right outer Join Syntax for **Oracle** is,

```
SELECT column-name-list FROM
table-name1, table-name2
ON table-name1.column-name(+) = table-name2.column-name;
```

Example of Right Outer Join

Once again the **class** table,

ID	NAME
1	abhi
2	adam
3	alex
4	anu
5	ashish

and the **class_info** table,

ID	Address
1	Kathmandu
2	Bhaktpur
3	Kalimati
7	Teku
8	Tripureshwor

Right Outer Join query will be,

```
SELECT * FROM class RIGHT OUTER JOIN class_info ON (class.id = class_info.id);
```

The resultant table will look like,

ID	NAME	ID	Address
1	abhi	1	Kathmandu
2	adam	2	Bhaktpur
3	alex	3	Kalimati
null	null	7	Teku
null	null	8	Tripureshwor

Full Outer Join

The full outer join returns a resultset table with the **matched data** of two table then remaining rows of both **left** table and then the **right** table.

Syntax of Full Outer Join is,

```
SELECT column-name-list FROM
table-name1 FULL OUTER JOIN table-name2
ON table-name1.column-name = table-name2.column-name;
```

Example of Full outer join is,

The **class** table,

ID	NAME
1	abhi
2	adam
3	alex
4	anu
5	ashish

and the **class_info** table,

ID	Address
1	Kathmandu
2	Bhaktpur
3	Kalimati
7	Teku
8	Tripureshwor

Full Outer Join query will be like,

```
SELECT * FROM class FULL OUTER JOIN class_info ON (class.id = class_info.id);
```

The resultset table will look like,

ID	NAME	ID	Address
1	abhi	1	Kathmandu
2	adam	2	Kupondole
3	alex	3	Kalimati
4	anu	null	null
5	ashish	null	null
null	null	7	Teku
null	null	8	Tripureshwor

SELF JOIN

It is used to join a table to itself as if the table were two tables; temporarily renaming at least one table in the SQL statement.

Syntax

The basic syntax of SELF JOIN is as follows −

```
SELECT a.column_name, b.column_name...
FROM table1 a, table1 b
WHERE a.common_field = b.common_field;
```

Here, the WHERE clause could be any given expression based on your requirement.

Example

Consider the following table.

CUSTOMERS Table is as follows.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Kathmandu	2000.00
2	Khilan	25	Kupondole	1500.00
3	Kaushik	23	Kalimati	2000.00
4	Chaitali	25	Kalanki	6500.00
5	Hardik	27	Teku	8500.00
6	Komal	22	Tripureshwor	4500.00
7	Muffy	24	Gaushala	10000.00

Now, let us join this table using SELF JOIN as follows −

```
SQL> SELECT  a.ID, b.NAME, a.SALARY
     FROM CUSTOMERS a, CUSTOMERS b
     WHERE a.SALARY < b.SALARY;
```

This would produce the following result −

ID	NAME	SALARY
2	Ramesh	1500.00
2	Kuashik	1500.00
1	Chaitali	2000.00
2	Chaitali	1500.00
3	Chaitali	2000.00
6	Chaitali	4500.00
1	Hardik	2000.00
2	Hardik	1500.00
3	Hardik	2000.00
4	Hardik	6500.00
6	Hardik	4500.00

1	Komal	2000.00
2	Komal	1500.00
3	Komal	2000.00
1	Muffy	2000.00
2	Muffy	1500.00
3	Muffy	2000.00
4	Muffy	6500.00
5	Muffy	8500.00
6	Muffy	4500.00

Embedded SQL

It is the one which combines the high level language with DB language like SQL. The high level languages which supports embedding SQLs within it are also known as host language. There are different host languages which supports embedding SQL within it are C, C++, C#, JAVA etc. In host language, a preprocessor converts the SQL statement into special API calls and then a regular compiler is used to compile the code.

Advantages

- It can create a more flexible, accessible interface for the user.
- It provides possible performance improvements.
- Database security improvement with access to application instead of individual users.

Structure of embedded sql

It defines step by step process of establishing a connection code in the DB within high level language.

Connect to DB

This can be done using CONNECT keyword that precede with "EXEC SQL" to indicate that it is a SQL statements.

Syntax: EXEC SQL CONNECT dbNAME;

Example : EXEC SQL CONNECT studentDB;

Declaration Section

To declare variables we use

BEGIN DECLARE & END DECLARE section

And these block should be enclosed within EXEC SQL and ';'

```
EXEC SQL BEGIN DECLARESECTION
        int std_ID,
        char std_NAME[15],
EXEC SQL END DECLARE SECTION;
```

For statement

We use

EXEC SQL Statements

Views

Views are the subset of tables. they also have set of records in the form of rows and columns. View can be created from one or many tables which depends on the written SQL query to create a view. Views are the type of virtual tables allows users to structure data in a way that users or classes of users find natural or intuitive. It also restrict access to the data in such a way that a user can see and modify exactly what they need. It also summarize data from various tables which can be used to generate report.

Create View

Syntax :
```
Create view view_name as
Select column1, column2, .....
from table_name
[Where condition]
```

Consider following **Sale** table,

oid	order_name	previous_balance	customer
11	ord1	2000	Alex
12	ord2	1000	Adam
13	ord3	2000	Abhi
14	ord4	1000	Adam
15	ord5	2000	Alex

SQL Query to Create a View from the above table will be,

```
CREATE or REPLACE VIEW sale_view
AS
SELECT * FROM Sale WHERE customer = 'Alex';
```

Displaying a VIEW

The syntax for displaying the data in a view is similar to fetching data from a table using a SELECTstatement.

```
SELECT * FROM sale_view;
```

Force VIEW Creation

FORCE keyword is used while creating a view, forcefully. This keyword is used to create a View even if the table does not exist. After creating a force View if we create the base table and enter values in it, the view will be automatically updated.

Syntax for forced View is,

```
CREATE or REPLACE FORCE VIEW view_name AS
    SELECT column_name(s)
    FROM table_name
    WHERE condition;
```

Update a VIEW

UPDATE command for view is same as for tables.

Syntax to Update a View is,

UPDATE view-name SET VALUE
WHERE condition;

NOTE: If we update a view it also updates base table data automatically.

Read-Only VIEW

We can create a view with read-only option to restrict access to the view.

Syntax to create a view with Read-Only Access

CREATE or REPLACE FORCE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition WITH read-only;

The above syntax will create view for **read-only** purpose, we cannot Update or Insert data into read-only view. It will throw an **error**.

Types of View

There are two types of view :

1. Simple view
2. Complex view

Simple View

It is created from single table. It can use DDL, DML directly. It does not contain functions and groups of data. It also does not include NOT NULL columns from base table.

Syntax for Simple View

Create View ViewName AS
Select column1,...columnN
From tableName;

Example :

We have table named orders:

oid	order_name	previous_balance	customer
11	ord1	2000	Alex
12	ord2	1000	Adam
13	ord3	2000	Abhi
14	ord4	1000	Adam
15	ord5	2000	Alex

Create View OrderView AS
Select oid, order_name
From orders;

oid	order_name
11	ord1
12	ord2
13	ord3
14	ord4
15	ord5

Complex View

It is created by using more than one tables. There should be relation between 2 tables to create complex view. It contains functions and group of data. DML operations could not always be performed through complex view. NOT NULL columns are included.

Example : Consider tables Employee(emp_name, emp_num, dept_code) and Department(dept_code, dept_name)

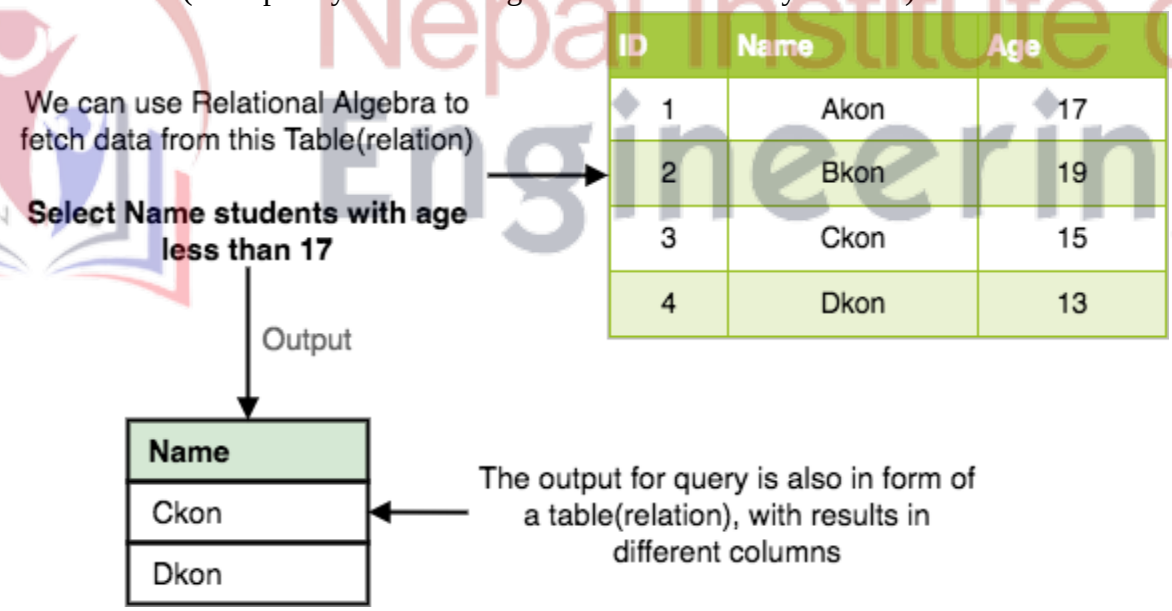
To create complex view:

```
Create View Emp_View AS
Select e. emp_name, d.dept_name
From Employee as e, Department as d
Where e.dept_code = d.dept_code;
```

Relational Algebra

Every database management system must define a query language to allow users to access the data stored in the database. **Relational Algebra** is a procedural query language used to query the database tables to access data in different ways.

In relational algebra, input is a relation(table from which data has to be accessed) and output is also a relation(a temporary table holding the data asked for by the user).



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.

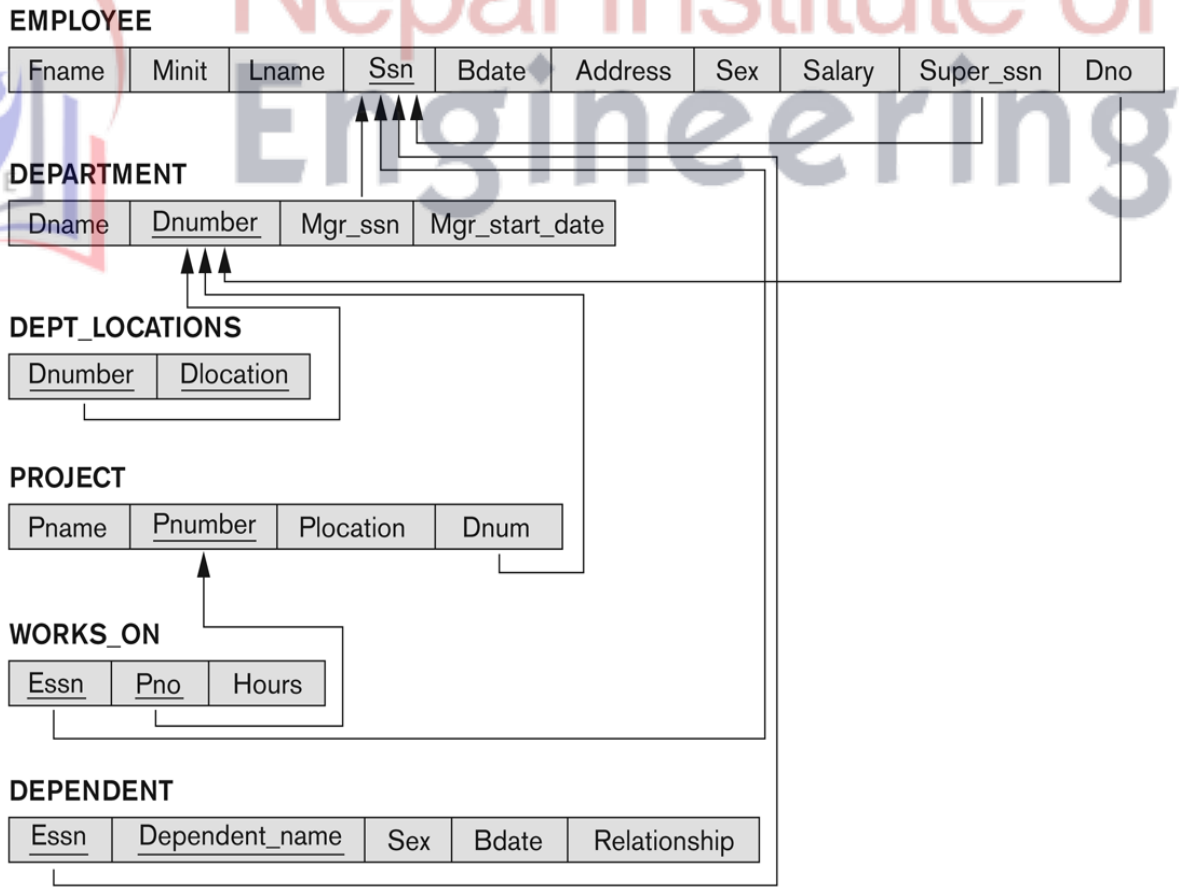
Relational Algebra consists of several groups of operations

- Unary Relational Operations
  - SELECT (symbol: σ (sigma))
  - PROJECT (symbol: π (pi))
  - RENAME (symbol: ρ (rho))
- Relational Algebra Operations From Set Theory
  - UNION (∪), INTERSECTION (∩), DIFFERENCE (or MINUS, –)
  - CARTESIAN PRODUCT (**x**)
- Binary Relational Operations
  - JOIN (several variations of JOIN exist)
  - DIVISION
- Additional Relational Operations
  - OUTER JOINS, OUTER UNION
  - AGGREGATE FUNCTIONS (These compute summary of information: for example, SUM, COUNT, AVG, MIN, MAX)

All examples are related to given database structure :

Figure 5.7

Referential integrity constraints displayed on the COMPANY relational database schema.



Unary Relational Operation

Select Operation (σ)

This is used to fetch rows(tuples) from table(relation) which satisfies a given condition.

Syntax: $\sigma_p(r)$

Where, σ represents the Select Predicate, r is the name of relation(table name in which you want to look for data), and p is the prepositional logic, where we specify the conditions that must be satisfied by the data. In prepositional logic, one can use **unary** and **binary** operators like =, <, > etc, to specify the conditions.

Examples:

Select the EMPLOYEE tuples whose department number is 4:

$\sigma_{DNO = 4}(EMPLOYEE)$

Select the employee tuples whose salary is greater than $30,000:

$\sigma_{SALARY > 30,000}(EMPLOYEE)$

In general, the *select* operation is denoted by $\sigma_{<selection\ condition>}(R)$ where

- the symbol σ (sigma) is used to denote the *select* operator
- the selection condition is a Boolean (conditional) expression specified on the attributes of relation R
- tuples that make the condition **true** are selected
  –appear in the result of the operation
- tuples that make the condition **false** are filtered out
  –discarded from the result of the operation

SELECT Operation Properties

- The SELECT operation $\sigma_{<selection\ condition>}(R)$ produces a relation S that has the same schema (same attributes) as R
- SELECT σ is commutative:

  $\sigma_{<condition1>}(\sigma_{<condition2>}(R)) = \sigma_{<condition2>}(\sigma_{<condition1>}(R))$
- Because of commutativity property, a cascade (sequence) of SELECT operations may be applied in any order:

  $\sigma_{<condition1>}(\sigma_{<condition2>}(\sigma_{<condition3>}(R)) = \sigma_{<condition2>}(\sigma_{<condition3>}(\sigma_{<condition1>}(R)))$
- A cascade of SELECT operations may be replaced by a single selection with a conjunction of all the conditions:

  $\sigma_{<condition1>}(\sigma_{<condition2>}(\sigma_{<condition3>}(R)) = \sigma_{<condition1>\Lambda<condition2>\Lambda<condition3>}(R)))$
- The number of tuples in the result of a SELECT is less than (or equal to) the number of tuples in the input relation R
- Example :

  $\sigma_{(Dno=4\ \Lambda\ Salary > 25000)\ \vee\ (Dno = 5\ \Lambda\ Salary > 30000)}$ (Employee)
- <selection condition> applied independently to each individual tuple *t* in *R*
  - If condition evaluates to TRUE, tuple selected
- Boolean conditions **AND**, **OR**, and **NOT**

- **Unary**
  - Applied to a single relation
- Selectivity
  - Fraction of tuples selected by a selection condition
- Select Operation commutative
- Cascade Select operations into a single operation with AND condition

Figure 5.6

One possible database state for the COMPANY relational database schema.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

Project Operation (∏)

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: $\prod_{A1, A2...}(r)$

where A1, A2 etc are attribute names(column names).

Example: To list each employee's first and last name and salary, the following is used:

$\prod_{LNAME,FNAME,SALARY}(EMPLOYEE)$

PROJECT Operation Properties

- The number of tuples in the result of projection □ <list>(R) is always less or equal to the number of tuples in R
    - If the list of attributes includes a *key* of R, then the number of tuples in the result of PROJECT is *equal* to the number of tuples in R
- PROJECT is *not* commutative
    - $\pi_{<list1>}(\pi_{<list2>}(R)) = \pi_{<list1>}(R)$ as long as <list2> contains the attributes in <list1>

Rename(ρ)

In some cases, we may want to rename the attributes of a relation or the relation name or both

- Useful when a query requires multiple operations
- Necessary in some cases (see JOIN operation later)
- The general RENAME operation □ can be expressed by any of the following forms:
- $\rho_{S(B1, B2, ..., Bn)}(R)$ changes both:
    - the relation name to S, *and*
    - the column (attribute) names to B1, B1, …..Bn
- $\rho_S(R)$ changes:
    - the *relation name* only to S
- $\rho_{(B1, B2, ..., Bn)}(R)$ changes:
    - the *column (attribute) names* only to B1, B1, …..Bn

For convenience, we also use a *shorthand* for renaming attributes in an intermediate relation:

- If we write:
    - RESULT <- $\pi_{FNAME, LNAME, SALARY}$ (DEP5_EMPS)
    - RESULT will have the *same attribute names* as DEP5_EMPS (same attributes as EMPLOYEE)
- If we write:
    - RESULT (F, M, L, S, B, A, SX, SAL, SU, DNO) <- $\rho_{RESULT(F.M.L.S.B,A,SX,SAL,SU,DNO)}$ (DEP5_EMPS)
    - The 10 attributes of DEP5_EMPS are *renamed* to F, M, L, S, B, A, SX, SAL, SU, DNO, respectively

Note : the symbol <- is an assignment operator.

Union Operation (∪)

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 autamatically eliminated from the result.

Syntax: A ∪ B

where A and B are relations.

For example, if we have two tables **RegularClass** and **ExtraClass**, both have a column **student** to save name of student, then,

$\prod_{Student}$(RegularClass) ∪ $\prod_{Student}$(ExtraClass)

Above operation will give us name of **Students** who are attending both regular classes and extra classes, eliminating repetition.

INTERSECTION

- denoted by ∩
- The result of the operation R ∩ S, is a relation that includes all tuples that are in both R and S
    - The attribute names in the result will be the same as the attribute names in R
- The two operand relations R and S must be "type compatible

Example : We have Depositor and Borrower Tables

Depositor

ID	NAME
1	A
2	B
3	C

Borrower

ID	NAME
2	B
3	A
5	D

Example : Find the customer name who has account in the bank as well as thay have taken the loan.

π_{NAME}(Depositor) ∩ π_{NAME}(Borrower)

Output :

NAME
A
B

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:

$\prod_{Student}(RegularClass) - \prod_{Student}(ExtraClass)$

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:

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

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

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: ρ(RelationNew, RelationOld)

Apart from these common operations Relational Algebra is also used for **Join** operations like,

- Natural Join
- Outer Join
- Theta join etc.

The relations used to understand extended operators are STUDENT, STUDENT_SPORTS, ALL_SPORTS and EMPLOYEE which are shown in Table 1, Table 2, Table 3 and Table 4 respectively.

STUDENT

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18
2	RAMESH	GURGAON	9652431543	18
3	SUJIT	ROHTAK	9156253131	20
4	SURESH	DELHI	9156768971	18

ALL_SPORTS

SPORTS
Badminton
Cricket

STUDENT_SPORTS

ROLL_NO	SPORTS
1	Badminton
2	Cricket
2	Badminton
4	Badminton

EMPLOYEE

EMP_NO	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18
5	NARESH	HISAR	9782918192	22
6	SWETA	RANCHI	9852617621	21
4	SURESH	DELHI	9156768971	18

Intersection (∩) : Intersection on two relations R1 and R2 can only be computed if R1 and R2 are **union compatible** (These two relation should have same number of attributes and corresponding attributes in two relations have same domain). Intersection operator when applied on two relations as R1∩R2 will give a relation with tuples which are in R1 as well as R2. Syntax:

Relation1 ∩ Relation2

Example: Find a person who is student as well as employee- **STUDENT ∩ EMPLOYEE**

In terms of basic operators (union and minus) :

STUDENT ∩ EMPLOYEE = STUDENT + EMPLOYEE - (STUDENT U EMPLOYEE)

RESULT:

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18
4	SURESH	DELHI	9156768971	18

Conditional Join($\bowtie_c$) : Conditional Join is used when you want to join two or more relation based on some conditions. Example: Select students whose ROLL_NO is greater than EMP_NO of employees

STUDENT $\bowtie_{c}$ STUDENT.ROLL_NO>EMPLOYEE.EMP_NO EMPLOYEE

In terms of basic operators (cross product and selection) :

σ (STUDENT.ROLL_NO>EMPLOYEE.EMP_NO)(STUDENT×EMPLOYEE)

RESULT:

ROLL_NO	NAME	ADDRESS	PHONE	AGE	EMP_NO	NAME	ADDRESS	PHONE	AGE
2	RAMESH	GURGAON	9652431543	18	1	RAM	DELHI	9455123451	18
3	SUJIT	ROHTAK	9156253131	20	1	RAM	DELHI	9455123451	18
4	SURESH	DELHI	9156768971	18	1	RAM	DELHI	9455123451	18

Equijoin(⋈) : Equijoin is a **special case of conditional join** where only equality condition holds between a pair of attributes. As values of two attributes will be equal in result of equijoin, only one attribute will be appeared in result.

Example: Select students whose ROLL_NO is equal to EMP_NO of employees

STUDENT⋈$_{STUDENT.ROLL_NO=EMPLOYEE.EMP_NO}$EMPLOYEE

In terms of basic operators (cross product, selection and projection) :

$\prod_{(STUDENT.ROLL\_NO,\ STUDENT.NAME,\ STUDENT.ADDRESS,\ STUDENT.PHONE,\ STUDENT.AGE\ EMPLOYEE.NAME,\ EMPLOYEE.ADDRESS,\ EMPLOYEE.PHONE,\ EMPLOYEE>AGE)}(\sigma_{(STUDENT.ROLL\_NO=EMPLOYEE.EMP\_NO)}(STUDENT \times EMPLOYEE))$

RESULT:

ROLL_NO	NAME	ADDRESS	PHONE	AGE	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18	RAM	DELHI	9455123451	18
4	SURESH	DELHI	9156768971	18	SURESH	DELHI	9156768971	18

Natural Join(⋈) : It is a special case of equijoin in which equality condition hold on all attributes which have same name in relations R and S (relations on which join operation is applied). While applying natural join on two relations, there is no need to write equality condition explicitly. Natural Join will also return the similar attributes only once as their value will be same in resulting relation.

Example: Select students whose ROLL_NO is equal to ROLL_NO of STUDENT_SPORTS as:

STUDENT⋈STUDENT_SPORTS

In terms of basic operators (cross product, selection and projection) :

$\prod_{(STUDENT.ROLL\_NO,\ STUDENT.NAME,\ STUDENT.ADDRESS,\ STUDENT.PHONE,\ STUDENT.AGE\ STUDENT\_SPORTS.SPORTS)}(\sigma_{(STUDENT.ROLL\_NO=STUDENT\_SPORTS.ROLL\_NO)}(STUDENT \times STUDENT\_SPORTS))$

RESULT:

ROLL_NO	NAME	ADDRESS	PHONE	AGE	SPORTS
1	RAM	DELHI	9455123451	18	Badminton
2	RAMESH	GURGAON	9652431543	18	Cricket
2	RAMESH	GURGAON	9652431543	18	Badminton
4	SURESH	DELHI	9156768971	18	Badminton

Natural Join is by default inner join because the tuples which does not satisfy the conditions of join does not appear in result set. e.g.; The tuple having ROLL_NO 3 in STUDENT does not match with any tuple in STUDENT_SPORTS, so it has not been a part of result set.

Left Outer Join(⟕) : When applying join on two relations R and S, some tuples of R or S does not appear in result set which does not satisfy the join conditions. But Left Outer Joins gives all tuples of R in the result set. The tuples of R which do not satisfy join condition will have values as NULL for attributes of S.

Example:Select students whose ROLL_NO is greater than EMP_NO of employees and details of other students as well

STUDENT⟕$_{STUDENT.ROLL_NO>EMPLOYEE.EMP_NO}$EMPLOYEE

RESULT

ROLL_NO	NAME	ADDRESS	PHONE	AGE	EMP_NO	NAME	ADDRESS	PHONE	AGE
2	RAMESH	GURGAON	9652431543	18	1	RAM	DELHI	9455123451	18
3	SUJIT	ROHTAK	9156253131	20	1	RAM	DELHI	9455123451	18
4	SURESH	DELHI	9156768971	18	1	RAM	DELHI	9455123451	18
1	RAM	DELHI	9455123451	18	NULL	NULL	NULL	NULL	NULL

Right Outer Join(⋈) : When applying join on two relations R and S, some tuples of R or S does not appear in result set which does not satisfy the join conditions. But Right Outer Joins gives all tuples of S in the result set. The tuples of S which do not satisfy join condition will have values as NULL for attributes of R.

Example: Select students whose ROLL_NO is greater than EMP_NO of employees and details of other Employees as well

STUDENT⋈$_{STUDENT.ROLL_NO>EMPLOYEE.EMP_NO}$EMPLOYEE

RESULT:

ROLL_NO	NAME	ADDRESS	PHONE	AGE	EMP_NO	NAME	ADDRESS	PHONE	AGE
2	RAMESH	GURGAON	9652431543	18	1	RAM	DELHI	9455123451	18
3	SUJIT	ROHTAK	9156253131	20	1	RAM	DELHI	9455123451	18
4	SURESH	DELHI	9156768971	18	1	RAM	DELHI	9455123451	18
NULL	NULL	NULL	NULL	NULL	5	NARESH	HISAR	9782918192	22
NULL	NULL	NULL	NULL	NULL	6	SWETA	RANCHI	9852617621	21
NULL	NULL	NULL	NULL	NULL	4	SURESH	DELHI	9156768971	18

Full Outer Join(⋈) : When applying join on two relations R and S, some tuples of R or S does not appear in result set which does not satisfy the join conditions. But Full Outer Joins gives all tuples of S and all tuples of R in the result set. The tuples of S which do not satisfy join condition will have values as NULL for attributes of R and vice versa.

Example:Select students whose ROLL_NO is greater than EMP_NO of employees and details of other Employees as well and other Students as well

STUDENT⋈$_{STUDENT.ROLL_NO>EMPLOYEE.EMP_NO}$EMPLOYEE

RESULT:

ROLL_NO	NAME	ADDRESS	PHONE	AGE	EMP_NO	NAME	ADDRESS	PHONE	AGE
2	RAMESH	GURGAON	9652431543	18	1	RAM	DELHI	9455123451	18
3	SUJIT	ROHTAK	9156253131	20	1	RAM	DELHI	9455123451	18
4	SURESH	DELHI	9156768971	18	1	RAM	DELHI	9455123451	18
NULL	NULL	NULL	NULL	NULL	5	NARESH	HISAR	9782918192	22
NULL	NULL	NULL	NULL	NULL	6	SWETA	RANCHI	9852617621	21
NULL	NULL	NULL	NULL	NULL	4	SURESH	DELHI	9156768971	18
1	RAM	DELHI	9455123451	18	NULL	NULL	NULL	NULL	NULL

Division Operator (÷) : Division operator A÷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.

Consider the relation STUDENT_SPORTS and ALL_SPORTS given in Table 2 and Table 3 above.

To apply division operator as

STUDENT_SPORTS÷ ALL_SPORTS

- The operation is valid as attributes in ALL_SPORTS is a proper subset of attributes in STUDENT_SPORTS.
- The attributes in resulting relation will have attributes {ROLL_NO,SPORTS}-{SPORTS}=ROLL_NO
- The tuples in resulting relation will have those ROLL_NO which are associated with all B's tuple {Badminton, Cricket}. ROLL_NO 1 and 4 are associated to Badminton only. ROLL_NO 2 is associated to all tuples of B. So the resulting relation will be:

ROLL_NO
2

Aggregate Function

Very useful to apply a function to a collection of values to generate a single result ¨

Most common aggregate functions:

- sum - sums the values in the collection
- avg - computes average of values in the collection
- count - counts number of elements in the collection
- min - returns minimum value in the collection
- max - returns maximum value in the collection ¨

Aggregate functions work on multisets, not sets

- A value can appear in the input multiple times

General Expression For Aggregate Functions in Relational Algebra is :-

$$_{<Grouping>}G_{<Functions>}(R)$$

whereas is a list of [MIN|MAX|AVERAGE|SUM|COUNT]

Usage Examples

$G_{MAX\ (Salary)}$ (EMPLOYEE) - Retrieves the maximum salary of an employee

$G_{MIN\ (Salary)}$ (EMPLOYEE) - Retrieves the minimum salary

$G_{SUM\ (Salary)}$ (EMPLOYEE) - Retrieves the sum of all salaries

$G_{COUNT\ (Ssn),\ AVERAGE\ (Salary)}$ (EMPLOYEE) - computes the count (number) of employees and their average salary

Relational Calculus

Contrary to Relational Algebra which is a procedural query language to fetch data and which also explains how it is done, **Relational Calculus** in non-procedural query language and has no description about how the query will work or the data will b fetched. It only focusses on what to do, and not on how to do it.

Relational Calculus exists in two forms:

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

Tuple Relational Calculus (TRC)

In tuple relational calculus, we work on filtering tuples based on the given condition.

Syntax: { T | Condition }

In this form of relational calculus, we define a tuple variable, specify the table(relation) name in which the tuple is to be searched for, along with a condition.

We can also specify column name using a . dot operator, with the tuple variable to only get a certain attribute(column) in result.

A lot of informtion, right! Give it some time to sink in.

A tuple variable is nothing but a name, can be anything, generally we use a single alphabet for this, so let's say T is a tuple variable.

To specify the name of the relation(table) in which we want to look for data, we do the following:

Relation(T), where T is our tuple variable.
For example if our table is **Student**, we would put it as Student(T)
Then comes the condition part, to specify a condition applicable for a particluar attribute(column), we can use the . dot variable with the tuple variable to specify it, like in table **Student**, if we want to get data for students with age greater than 17, then, we can write it as,
T.age > 17, where T is our tuple variable.
Putting it all together, if we want to use Tuple Relational Calculus to fetch names of students, from table **Student**, with age greater than **17**, then, for T being our tuple variable,
T.name | Student(T) AND T.age > 17

Domain Relational Calculus (DRC)

In domain relational calculus, filtering is done based on the domain of the attributes and not based on the tuple values.
Syntax: { c1, c2, c3, ..., cn | F(c1, c2, c3, ... ,cn)}
where, c1, c2... etc represents domain of attributes(columns) and F defines the formula including the condition for fetching the data.
For example,
{< name, age > | ∈ Student ∧ age > 17}
Again, the above query will return the names and ages of the students in the table **Student** who are older than 17.