**Chapter 6**

**Structure Query Language (SQL)**

1. **Introduction SQL**
2. **Data Definition Language (DDL)**
3. **Data Manipulation Language ( DML)**
4. **Data Control Language (DCL)**

**Structured Query Language(SQL)**

**6.1 Introduction**

Structured Query Language (SQL) is a standard computer language for relational database management and data manipulation. SQL is used to query, insert, update and modify data. Most relational databases support SQL, which is an added benefit for database administrators (DBAs), as they are often required to support databases across several different platforms.

First developed in the early 1970s at IBM by Raymond Boyce and Donald Chamberlin, SQL was commercially released by Relational Software Inc. (now known as Oracle Corporation) in 1979. The current standard SQL version is voluntary, vendor-compliant and monitored by the American National Standards Institute (ANSI). Most major vendors also have proprietary versions that are incorporated and built on ANSI SQL, e.g., SQL\*Plus (Oracle), and Transact-SQL (T-SQL) (Microsoft).

One of the most fundamental DBA rites of passage is learning SQL, which begins with writing the first SELECT statement or SQL script without a graphical user interfaces (GUI). Increasingly, relational databases use GUIs for easier database management, and queries can now be simplified with graphical tools, e.g., drag-and-drop wizards. However, learning SQL is imperative because such tools are never as powerful as SQL.

**6.2 What can SQL do?**

1. SQL can execute queries against a database .
2. SQL can retrieve data from a database .
3. SQL can insert records in a database .
4. SQL can update records in a database.
5. SQL can delete records from a database.
6. SQL can create new databases .
7. SQL can create new tables in a database.
8. SQL can create stored procedures in a database.
9. SQL can create views in a database .

10. SQL can set permissions on tables, procedures, and views .

**6.3 Database Languages**

A database system provides a **data definition language** to specify the database schema and a **data manipulation language** to express database queries and updates and a **data control language** to configure security access to relational databases . In practice, the data definition and data manipulation languages are not two separate languages; instead they simply form parts of a single database language, such as the widely used SQL language.

* + 1. **Data Definition Language (DDL)**

The SQL DDL allows specification of not only a set of relations, but also information about each relation, including

*•* The schema for each relation

*•* The domain of values associated with each attribute

*•* The integrity constraints

*•* The set of indices to be maintained for each relation

*•* The security and authorization information for each relation

*•* The physical storage structure of each relation on disk

Data Definition Language (DDL): statements are used to define the database structure or schema. Some examples:

* CREATE - to create objects in the database.
* ALTER - alters the structure of the database.
* DROP - delete objects from the database.
* TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed.
* COMMENT - add comments to the data dictionary.
* RENAME - rename an object.
  + 1. **Data Manipulation Language(DML)**

A **data-manipulation language (DML)** is a language that enables users to access or manipulate data as organized by the appropriate data model.

**Data manipulation** is

*•* The retrieval of information stored in the database

*•* The insertion of new information into the database

*•* The deletion of information from the database

*•* The modification of information stored in the database

There are basically two types:

*•* **Procedural DMLs** require a user to specify *what* data are needed and *how* to get those data.

*•* **Declarative DMLs** (also referred to as **nonprocedural** DMLs) require a user to specify *what* data are needed *without* specifying how to get those data.

Declarative DMLs are usually easier to learn and use than are procedural DMLs.

However, since a user does not have to specify how to get the data, the database system has to figure out an efficient means of accessing data. The DML component of the SQL language is nonprocedural.

A **query** is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a **query language**. Although technically incorrect, it is common practice to use the terms *query language* and *data manipulation*

*language* synonymously.

Data Manipulation Language (DML) statements are used for managing data within schema objects. Some examples:

* SELECT - retrieve data from the a database.
* INSERT - insert data into a table.
* UPDATE - updates existing data within a table.
* DELETE - deletes all records from a table, the space for the records remain.
* MERGE - UPSERT operation (insert or update).
* CALL - call a PL/SQL or Java subprogram.
* EXPLAIN PLAN - explain access path to data.
* LOCK TABLE - control concurrency.
  + 1. **Data Control Language:**

Data Control Language (DCL) :statement is a subset of the Structured Query Language (SQL) that allows database administrators to configure security access to relational databases. Some examples:

* GRANT - gives user's access privileges to database.
* REVOKE - withdraw access privileges given with the GRANT command.

# 6.4 SQL - Data Types

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

You would use these data types while creating your tables. You would choose a particular data type for a table column based on your requirement.

The SQL standard supports a variety of built-in domain types, including:

*•* **char**(*n*): A fixed-length character string with user-specified length *n*. The full form, **character**, can be used instead.

*•* **varchar**(*n*): A variable-length character string with user-specified maximum length *n*. The full form, **character varying**, is equivalent.

*•* **int**: An integer (a finite subset of the integers that is machine dependent). The full form, **integer**, is equivalent.

*•* **smallint**: A small integer (a machine-dependent subset of the integer domain type).

*•* **numeric**(*p, d*): A fixed-point number with user-specified precision. The number consists of *p* digits (plus a sign), and *d* of the *p* digits are to the right of the decimal point. Thus, **numeric**(3,1) allows 44*.*5 to be stored exactly, but neither 444*.*5 or 0*.*32 can be stored exactly in a field of this type.

*•* **real, double precision**: Floating-point and double-precision floating-point numbers with machine-dependent precision.

*•* **float**(*n*): A floating-point number, with precision of at least *n* digits.

*•* **date**: A calendar date containing a (four-digit) year, month, and day of the month.

*•* **time**: The time of day, in hours, minutes, and seconds. A variant, **time**(*p*), can be used to specify the number of fractional digits for seconds (the default being 0). It is also possible to store time zone information along with the time.

*•* **timestamp**: A combination of **date** and **time**. A variant, **timestamp**(*p*), can be used to specify the number of fractional digits for seconds (the default here being 6).

Date and time values can be specified like this:

**date** ’2001-04-25’

**time** ’09:30:00’

**timestamp** ’2001-04-25 10:29:01.45’

Dates must be specified in the format year followed by month followed by day, as shown. The seconds field of **time** or **timestamp** can have a fractional part, as in the timestamp above. We can use an expression of the form **cast** *e* **as** *t* to convert a character string (or string valued expression) *e* to the type *t*, where *t* is one of **date, time**,

or **timestamp**. The string must be in the appropriate format as illustrated at the beginning of this paragraph.

To extract individual fields of a **date** or **time** value *d*, we can use **extract** (*field* **from** *d*), where *field* can be one of **year, month, day, hour, minute**, or **second.**

**6.4.1 SQL Data Type Quick Reference**

However, different databases offer different choices for the data type definition.

The following table shows some of the common names of data types between the various database platforms:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Data type** | **Access** | **SQL Server** | **Oracle** | **MySQL** | **PostgreSQL** |
| *Boolean* | Yes/No | Bit | Byte | N/A | Boolean |
| *Integer* | Number (integer) | Int | Number | Int Integer | Int Integer |
| *Float* | Number (single) | Float Real | Number | Float | Numeric |
| *Currency* | Currency | Money | N/A | N/A | Money |
| *string (fixed)* | N/A | Char | Char | Char | Char |
| *string (variable)* | Text (<256) Memo (65k+) | Varchar | Varchar Varchar2 | Varchar | Varchar |
| *binary object* | OLE Object Memo | Binary (fixed up to 8K) Varbinary (<8K) Image (<2GB) | Long Raw | Blob Text | Binary Varbinary |

**Note:** Data types might have different names in different database. And even if the name is the same, the size and other details may be different! **Always check the documentation!**

### 6.5 Data Definition Language (DDL) command

### 6.5.1. Create Command

**create** is a DDL command used to create a table or a database.

#### 6.5.1.1 Creating a Database

To create a database in RDBMS, *create* command is uses. Following is the Syntax,

**create** database *database-name*;

#### Example for Creating Database

create database Test;

The above command will create a database named **Test**.

#### 6.5.1.2 Creating a Table

*create* command is also used to create a table. We can specify names and data types of various columns along. 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 given table name and column information.

#### Example for creating Table

create table Student(id int, name varchar, age int);

The above command will create a new table **Student** in database system with 3 columns, namely id, name and age.

**6.5.2 Alter command**

*alter* command is used for alteration of table structures. There are various uses of *alter* command, such as,

* to add a column to existing table
* to rename any existing column
* to change data type of any column or to modify its size.
* *alter* is also used to drop a column.

**6.5.2.1 To Add Column to existing Table**

Using alter command we can add a column to an 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 char);

The above command will add a new column *address* to the **Student** table

**6.5.2.2 To Add Multiple Column to existing Table**

Using alter command we can even add multiple columns to an 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

**6.5.2.3 To Add column with Default Value**

alter command can add a new column to an existing table with default values. Following is the Syntax,

**alter** table *table-name* add(**column-name1** *datatype1* **default** *data*);

Here is an Example for this,

alter table Student add(dob date default '1-Jan-99');

The above command will add a new column with default value to the **Student** table

**6.5.2.4 To Modify an existing Column**

alter command is used to modify data type of an 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(30));

The above command will modify *address* column of the **Student table**

**6.5.2.5 To 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 column-name;

Here is an Example for this,

alter table Student rename address to Location;

The above command will rename *address* column to *Location*.

**6.5.2.6 To Drop a Column**

alter command is also used to drop columns also. 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 *address* column from the **Student table**

#### 6.5.3 Truncate Command

*truncate* command removes all records from a table. But this command will not destroy the table's structure. When we apply truncate command on a table its Primary key is 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 of **Student** table.

**truncate** command is different from **delete** command. delete command will delete all the rows from a table whereas truncate command re-initializes a table(like a newly created table).

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

#### 6.5.4 Drop command

*drop* query completely removes a table from database. This command will also destroy the table structure. 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. For Example, to drop a database,

drop database Test;

The above query will drop a database named **Test** from the system.

#### 6.5.5 Rename query

*rename* command is used to rename a table. Following is its Syntax,

**rename** table *old-table-name* to *new-table-name*

Here is an Example explaining it.

rename table Student to Student-record;

The above query will rename **Student** table to **Student-record**.

### 6.6 Data Manipulation Language( DML) 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.

#### 6.6.1) INSERT command

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

**INSERT** into *table-name* values(data1,data2,..)

Lets see an example,

Consider a table **Student** with following fields.

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |

INSERT into Student values(101,'Adam',15);

The above command will insert a record into **Student** table.

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |

#### Example to 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 value other column is set to null.

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex |  |

#### Example to 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 **age** column of student table has 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')

#### 6.6.2) UPDATE command

Update command is used to update a row of a table. Following is its general syntax,

**UPDATE** *table-name* set column-name = value *where* **condition**;

Let's see an example,

update Student set age=18 where s\_id=102;

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Chris | 14 |

#### Example to Update multiple columns

UPDATE Student set s\_name='Abhi',age=17 where s\_id=103;

The above command will update two columns of a record.

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |

#### 6.6.3) Delete command

Delete command is used to delete data from a table. Delete command can also be used with condition to delete a particular row. Following is its general syntax,

**DELETE** from *table-name*;

#### Example to Delete all Records from a Table

DELETE from Student;

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

#### Example to Delete a particular Record from a Table

Consider the following **Student** table

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |

DELETE from Student where s\_id=103;

The above command will delete the record where s\_id is 103 from **Student** table.

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |

### 6.7 WHERE clause

Where clause is used to specify condition while retrieving data from table. *Where* clause is used mostly with *Select*, *Update* and *Delete* query. If condition specified by *where* clause is true then only the result from table is returned.

#### Syntax for WHERE clause

*SELECT* column-name1,

column-name2,

column-name3,

column-nameN

from table-name **WHERE [condition]**;

#### Example using WHERE clause

Consider a **Student** table,

|  |  |  |  |
| --- | --- | --- | --- |
| **s\_id** | **s\_Name** | **Age** | **address** |
| 101 | Adam | 15 | Noida |
| 102 | Alex | 18 | Delhi |
| 103 | Abhi | 17 | Rohtak |
| 104 | Ankit | 22 | Panipat |

Now we will use a SELECT statement to display data of the table, based on a condition, which we will add to the SELECT query using WHERE clause.

SELECT s\_id,

s\_name,

age,

address

from Student **WHERE** s\_id=101;

|  |  |  |  |
| --- | --- | --- | --- |
| **s\_id** | **s\_Name** | **Age** | **address** |
| 101 | Adam | 15 | Noida |

### 6.8 SELECT Query

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

#### Syntax of SELECT Query

**SELECT** column-name1, column-name2, column-name3, column-nameN from *table-name*;

#### Example for SELECT Query

Conside the following **Student** table,

|  |  |  |  |
| --- | --- | --- | --- |
| **S\_id** | **S\_Name** | **Age** | **address** |
| 101 | Adam | 15 | Noida |
| 102 | Alex | 18 | Delhi |
| 103 | Abhi | 17 | Rohtak |
| 104 | Ankit | 22 | Panipat |

SELECT s\_id, s\_name, age from Student.

The above query will fetch information of s\_id, s\_name and age column from Student table

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **Age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |
| 104 | Ankit | 22 |

#### Example to Select all Records from 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.

SELECT \* from student;

The above query will show all the records of Student table, that means it will show complete Student table as result.

|  |  |  |  |
| --- | --- | --- | --- |
| **S\_id** | **S\_Name** | **Age** | **address** |
| 101 | Adam | 15 | Noida |
| 102 | Alex | 18 | Delhi |
| 103 | Abhi | 17 | Rohtak |
| 104 | Ankit | 22 | Panipat |

#### Example to Select particular Record based on Condition

SELECT \* from Student **WHERE** s\_name = 'Abhi';

|  |  |  |  |
| --- | --- | --- | --- |
| 103 | Abhi | 17 | Rohtak |

#### Example to Perform Simple Calculations using Select Query

Conside the following **Employee** table.

|  |  |  |  |
| --- | --- | --- | --- |
| **Eid** | **Name** | **Age** | **Salary** |
| 101 | Adam | 26 | 5000 |
| 102 | Ricky | 42 | 8000 |
| 103 | Abhi | 22 | 10000 |
| 104 | Rohan | 35 | 5000 |

SELECT eid, name, salary+3000 from Employee;

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

|  |  |  |
| --- | --- | --- |
| **Eid** | **Name** | **salary+3000** |
| 101 | Adam | 8000 |
| 102 | Ricky | 11000 |
| 103 | Abhi | 13000 |
| 104 | Rohan | 8000 |

**6.9 Like clause**

**Like** clause is used as condition in SQL query. **Like** clause compares data with an expression using wildcard operators. It is used to find similar data from the table.

**Wildcard operators**

There are two wildcard operators that are used in like clause.

* **Percent sign %** : represents zero, one or more than one character.
* **Underscore sign \_** : represents only one character.

**Example of LIKE clause**

Consider the following **Student** table.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **s\_Name** | **Age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |

SELECT \* from Student where s\_name like 'A%';

The above query will return all records where **s\_name** starts with character 'A'.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **s\_Name** | **Age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |

**Example:**

SELECT \* from Student where s\_name like '\_d%';

The above query will return all records from **Student** table where **s\_name** contain 'd' as second character.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **s\_Name** | **Age** |
| 101 | Adam | 15 |

**Example:**

SELECT \* from Student where s\_name like '%x';

The above query will return all records from **Student** table where **s\_name** contain 'x' as last character.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **s\_Name** | **Age** |
| 102 | Alex | 18 |

### 6.9 Order By Clause

Order by clause is used with **Select** statement for arranging retrieved data in sorted order. The **Order by**clause by default sort data in ascending order. To sort data in descending order **DESC** keyword is used with**Order by** clause.

#### Syntax of Order By

*SELECT* column-list|\* from table-name **order by** *asc*|*desc*;

#### Example using Order by

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 |

SELECT \* from Emp **order by** salary;

The above query will return result in ascending order of the **salary**.

|  |  |  |  |
| --- | --- | --- | --- |
| **Eid** | **Name** | **Age** | **salary** |
| 403 | Rohan | 34 | 6000 |
| 402 | Shane | 29 | 8000 |
| 405 | Tiger | 35 | 8000 |
| 401 | Anu | 22 | 9000 |
| 404 | Scott | 44 | 10000 |

#### Example of Order by DESC

Consider the **Emp** table described above,

SELECT \* from Emp order by salary DESC;

The above query will return result in descending order of the **salary**.

|  |  |  |  |
| --- | --- | --- | --- |
| **Eid** | **Name** | **age** | **Salary** |
| 404 | Scott | 44 | 10000 |
| 401 | Anu | 22 | 9000 |
| 405 | Tiger | 35 | 8000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |

### 6.10 HAVING Clause

having clause is used with SQL Queries to give more precise condition for a statement. It is used to mention condition in Group based SQL functions, just like WHERE clause.

Syntax for having will be,

select column\_name, function(column\_name)

FROM table\_name

WHERE column\_name condition

GROUP BY column\_name

**HAVING** function(column\_name) condition

#### Example of HAVING Statement

Consider the 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 |

Suppose we want to find the customer whose previous\_balance sum is more than 3000.

We will use the below SQL query,

SELECT \*

from sale group customer

having sum(previous\_balance) > 3000

Result will be,

|  |  |  |  |
| --- | --- | --- | --- |
| **oid** | **order\_name** | **previous\_balance** | **customer** |
| 11 | ord1 | 2000 | Alex |

### 6.11 Distinct keyword

The **distinct** keyword is used with **Select** statement to retrieve unique values from the table. **Distinct** removes all the duplicate records while retrieving from database.

#### Syntax for DISTINCT Keyword

**SELECT** *distinct* column-name from *table-name*;

#### Example

Consider the following **Emp** table.

|  |  |  |  |
| --- | --- | --- | --- |
| **Eid** | **Name** | **Age** | **Salary** |
| 401 | Anu | 22 | 5000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 10000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

select distinct salary from Emp;

The above query will return only the unique salary from **Emp** table

|  |
| --- |
| **salary** |
| 5000 |
| 8000 |
| 10000 |

### 6.12 AND & OR operator

**AND** and **OR** operators are used with **Where** clause to make more precise conditions for fetching data from database by combining more than one condition together.

#### 6.12.1 AND operator

AND operator is used to set multiple conditions with *Where* clause.

#### Example of AND

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **Eid** | **Name** | **Age** | **Salary** |
| 401 | Anu | 22 | 5000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 12000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 9000 |

SELECT \* from Emp WHERE salary < 10000 **AND** age > 25

The above query will return records where salary is less than 10000 and age greater than 25.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Eid** | **Name** | | **Age** | **Salary** |
| 402 | Shane | 29 | | 8000 |
| 405 | Tiger | 35 | | 9000 |

#### 6.12.2 OR operator

OR operator is also used to combine multiple conditions with *Where* clause. The only difference between AND and OR is their behavior. When we use AND to combine two or more than two conditions, records satisfying all the condition will be in the result. But in case of OR, at least one condition from the conditions specified must be satisfied by any record to be in the result.

#### Example of OR

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **Eid** | **Name** | **Age** | **Salary** |
| 401 | Anu | 22 | 5000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 12000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 9000 |

SELECT \* from Emp WHERE salary > 10000 **OR** age > 25

The above query will return records where either salary is greater than 10000 or age greater than 25.

|  |  |  |  |
| --- | --- | --- | --- |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 12000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 9000 |

**6.13 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 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

**6.13.1 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 NOT NULL constraint is that it cannot be defined at table level.

**Example using NOT NULL constraint**

CREATE table Student(s\_id int NOT NULL, Name varchar(60), Age int);

The above query will declare that the **s\_id** field of **Student** table will not take NULL value.

**6.13.2 UNIQUE Constraint**

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

**Example using UNIQUE constraint when creating a Table (Table Level)**

CREATE table Student(s\_id int NOT NULL UNIQUE, Name varchar(60), Age int);

The above query will declare that the **s\_id** field of **Student** table will only have unique values and won't take NULL value.

**Example using UNIQUE constraint after Table is created (Column Level)**

ALTER table Student add UNIQUE(s\_id);

The above query specifies that **s\_id** field of **Student** table will only have unique value.

**6.13.3 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.

**Example using PRIMARY KEY constraint at Table Level**

CREATE table Student (s\_id int **PRIMARY KEY**, Name varchar(60) NOT NULL, Age int);

The above command will creates a PRIMARY KEY on the s\_id.

**Example using PRIMARY KEY constraint at Column Level**

ALTER table Student add PRIMARY KEY (s\_id);

The above command will creates a PRIMARY KEY on the s\_id.

**6.13.4 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 it using two table.

**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.

**Example 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*));

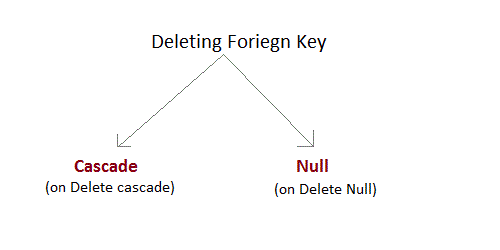
In this query, c\_id in table Order\_Detail is made as foriegn key, which is a reference of c\_id column of Customer\_Detail.

**Example using FOREIGN KEY constraint at Column Level**

ALTER table Order\_Detail add **FOREIGN KEY** (c\_id) REFERENCES Customer\_Detail(c\_id);

**Behavior of Foreign Key Column on Delete**

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



* **On Delete Cascade :** This will remove the record from child table, if that value of foreign key is deleted from the main table.
* **On Delete Null :** This will set all the values in that record of child table as NULL, for which the value of foreign key is selected from the main table.
* 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

**6.13.4 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 using CHECK constraint at Table Level**

create table Student(s\_id int NOT NULL **CHECK(s\_id > 0)**,

Name varchar(60) NOT NULL,

Age int);

The above query will restrict the s\_id value to be greater than zero.

**Example using CHECK constraint at Column Level**

ALTER table Student add CHECK(s\_id > 0);

**6.14 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,

* Aggregate Functions
* Scalar Functions

**6.14.1 Aggregate Functions**

These functions return a single value after calculating from a group of values. Following are some frequently used Aggregate functions.

**1) AVG()**

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

Its general Syntax is,

SELECT **AVG**(column\_name) from *table\_name*

**Example using AVG()**

Consider 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 of salary will be,

SELECT **avg(salary)** from Emp;

Result of the above query will be,

|  |
| --- |
| **avg(salary)** |
| 8200 |

**2) COUNT()**

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*

**Example using COUNT()**

Consider 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 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 |

**3) FIRST()**

First function returns first value of a selected column

Syntax for FIRST function is,

SELECT **FIRST**(column\_name) from *table-name*

**Example of FIRST()**

Consider 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

SELECT FIRST(salary) from Emp;

Result will be,

|  |
| --- |
| **first(salary)** |
| 9000 |

**4) LAST()**

LAST return the return last value from selected column

Syntax of LAST function is,

SELECT **LAST**(column\_name) from *table-name*

**Example of LAST()**

Consider 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 |

**5) MAX()**

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

Syntax of MAX function is,

SELECT **MAX**(column\_name) from *table-name*

**Example of MAX()**

Consider 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 Maximum salary is,

SELECT MAX(salary) from emp;

Result of the above query will be,

|  |
| --- |
| **MAX(salary)** |
| 10000 |

**6) MIN()**

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

Syntax for MIN function is,

SELECT **MIN**(column\_name) from *table-name*

**Example of MIN()**

Consider 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)** |
| 8000 |

**7) SUM()**

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

Syntax for SUM is,

SELECT SUM(column\_name) from *table-name*

**Example of SUM()**

Consider 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 |

**6.14.2 Scalar Functions**

Scalar functions return a single value from an input value. Following are some frequently used Scalar Functions.

**1) UCASE()**

UCASE function is used to convert value of string column to Uppercase character.

Syntax of UCASE,

SELECT **UCASE**(column\_name) from *table-name*

**Example of UCASE()**

Consider 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 for using UCASE is,

SELECT UCASE(name) from emp;

Result is,

|  |
| --- |
| **UCASE(name)** |
| ANU |
| SHANE |
| ROHAN |
| SCOTT |
| TIGER |

**2) LCASE()**

LCASE function is used to convert value of string column to Lowecase character.

Syntax for LCASE is,

SELECT **LCASE**(column\_name) from *table-name*

**Example of LCASE()**

Consider 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 for converting string value to Lower case is,

SELECT LCASE(name) from emp;

Result will be,

|  |
| --- |
| **LCASE(name)** |
| anu |
| shane |
| rohan |
| scott |
| tiger |

**3) MID()**

MID function is used to extract substrings from column values of string type in a table.

Syntax for MID function is,

SELECT **MID**(column\_name, start, length) from *table-name*

**Example of MID()**

|  |  |  |  |
| --- | --- | --- | --- |
| **Eid** | **Name** | **Age** | **Salary** |
| 401 | anu | 22 | 9000 |
| 402 | shane | 29 | 8000 |
| 403 | rohan | 34 | 6000 |
| 404 | scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

Consider following **Emp** table

SQL query will be,

select MID(name,2,2) from emp;

Result will come out to be,

|  |
| --- |
| **MID(name,2,2)** |
| Nu |
| ha |
| oh |
| co |
| ig |

**4) ROUND()**

ROUND function is used to round a numeric field to number of nearest integer. It is used on Decimal point values. Syntax of Round function is,

SELECT **ROUND**(column\_name, decimals) from *table-name*

**Example of ROUND()**

Consider following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **Eid** | **Name** | **Age** | **Salary** |
| 401 | anu | 22 | 9000.67 |
| 402 | shane | 29 | 8000.98 |
| 403 | rohan | 34 | 6000.45 |
| 404 | scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000.01 |

SQL query is,

SELECT ROUND(salary) from emp;

Result will be,

|  |
| --- |
| **ROUND(salary)** |
| 9001 |
| 8001 |
| 6000 |
| 10000 |
| 8000 |

**6.15 Join in SQL**

SQL Join is used to fetch data from two or more tables, which is joined to appear as single set of data. SQL Join is used for combining column from two or more tables by using values common to both tables. **Join** Keyword is used in SQL queries for joining two or more tables. Minimum required condition for joining table, is**(n-1)** where **n**, is number of tables. A table can also join to itself known as, **Self Join**.

**Types of Join**

The following are the types of JOIN that we can use in SQL.

* Inner
* Outer
* Left
* Right

**6.15.1 Cross JOIN or Cartesian Product**

This type of JOIN returns the Cartesian product of rows of 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**

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 4 | alex |

The **class** table,

The **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |

**Cross** JOIN query will be,

SELECT \*

from class,

cross JOIN class\_info;

The result table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 1 | DELHI |
| 4 | alex | 1 | DELHI |
| 1 | abhi | 2 | MUMBAI |
| 2 | adam | 2 | MUMBAI |
| 4 | alex | 2 | MUMBAI |
| 1 | abhi | 3 | CHENNAI |
| 2 | adam | 3 | CHENNAI |
| 4 | alex | 3 | CHENNAI |

**6.15.2 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 query.

Inner Join Syntax is,

SELECT column-name-list

from *table-name1*

**INNER JOIN**

*table-name2*

WHERE table-name1.column-name = table-name2.column-name;

**Example of Inner JOIN**

The **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |

The **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |

**Inner** JOIN query will be,

SELECT \* from class, class\_info where class.id = class\_info.id;

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |

The result table will look like,

**6.15.3 Natural JOIN**

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

Natural Join Syntax is,

SELECT \*

from *table-name1*

**NATURAL JOIN**

*table-name2*;

**Example of Natural JOIN**

The **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |

The **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |

**Natural join query will be,**

SELECT \* from class NATURAL JOIN class\_info;

The result table will look like,

|  |  |  |
| --- | --- | --- |
| **ID** | **NAME** | **Address** |
| 1 | abhi | DELHI |
| 2 | adam | MUMBAI |
| 3 | alex | CHENNAI |

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.

**6.15.5 Outer JOIN**

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

* Left Outer Join
* Right Outer Join
* Full Outer Join

**6.15.5.1 Left Outer Join**

The left outer join returns a result table with the **matched data** of two tables then remaining rows of the **left**table and null for the **right** table's column.

Left Outer Join syntax is,

SELECT column-name-list

from *table-name1*

**LEFT OUTER JOIN**

*table-name2*

on table-name1.column-name = table-name2.column-name;

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**

The **class** table,

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

The **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |
| 7 | NOIDA |
| 8 | PANIPAT |

**Left Outer Join** query will be,

SELECT \* FROM class LEFT OUTER JOIN class\_info ON (class.id=class\_info.id);

The result table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |
| 4 | anu | null | null |
| 5 | ashish | null | null |

**6.15.5.2 Right Outer Join**

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

Right Outer Join Syntax 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**

The **class** table,

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

The **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |
| 7 | NOIDA |
| 8 | PANIPAT |

**Right Outer Join** query will be,

SELECT \* FROM class RIGHT OUTER JOIN class\_info on (class.id=class\_info.id);

The result table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |
| null | null | 7 | NOIDA |
| null | null | 8 | PANIPAT |

**6.15.3 Full Outer Join**

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

Full Outer Join Syntax 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 |

The **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |
| 7 | NOIDA |
| 8 | PANIPAT |

**Full Outer Join** query will be like,

SELECT \* FROM class FULL OUTER JOIN class\_info on (class.id=class\_info.id);

The result table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |
| 4 | anu | null | null |
| 5 | ashish | null | null |
| Null | null | 7 | NOIDA |
| Null | null | 8 | PANIPAT |

### 6.16 SQL Alias

Alias is used to give an alias name to a table or a column. This is quite useful in case of large or complex queries. Alias is mainly used for giving a short alias name for a column or a table with complex names.

Syntax of Alias for table names,

**SELECT** column-name

from *table-name*

as **alias-name**

Following is an Example using Alias,

SELECT \* from Employee\_detail as **ed**;

Alias syntax for columns will be like,

**SELECT**

*column-name* as **alias-name**

from*table-name*

|  |  |
| --- | --- |
| **ID** | **Name** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |
| 5 | ashish |

Example using alias for columns,

SELECT customer\_id as **cid** from Emp;

#### Example of Alias in SQL Query

Consider the following two tables,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |
| 7 | NOIDA |
| 8 | PANIPAT |

The **class** table,The **class\_info** table,

Below is the Query to fetch data from both the tables using SQL Alias,

SELECT C.id, C.Name, Ci.Address from Class as C, Class\_info as Ci where C.id=Ci.id;

Result table look like,

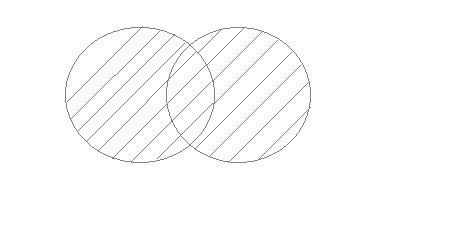
|  |  |  |
| --- | --- | --- |
| **ID** | **Name** | **Address** |
| 1 | abhi | DELHI |
| 2 | adam | MUMBAI |
| 3 | alex | CHENNAI |

### 6.17 Set Operation in SQL

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

#### 6.17.1 Union

UNION is used to combine the results of two or more Select statements. However it will eliminate duplicate rows from its result set. In case of union, number of columns and data type must be same in both the tables.



#### 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**

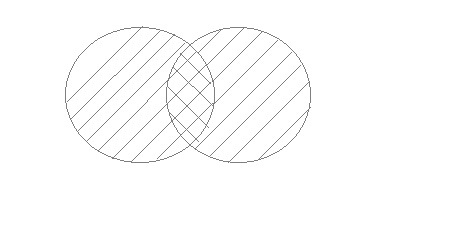
select \* from second

The result 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**

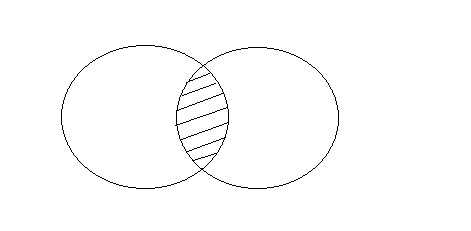
select \* from second

The result table will look like,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 2 | adam |
| 3 | Chester |

#### 6.17. 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. 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**

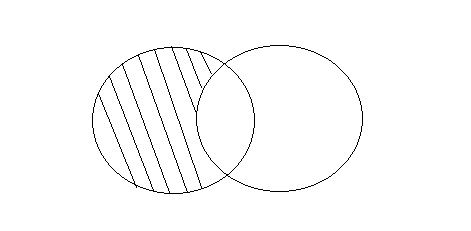
select \* from second

The result table will look like

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 2 | adam |

#### 6.17.4 Minus

Minus operation combines result of two Select statements and return only those result which belongs to first set of result. MySQL does not support INTERSECT operator.



#### 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 table will look like,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | Abhi |

### 6.18 SQL Sequence

Sequence is a feature supported by some database systems to produce unique values on demand. Some DBMS like **MySQL** supports AUTO\_INCREMENT in place of Sequence. AUTO\_INCREMENT is applied on columns, it automatically increments the column value by 1 each time a new record is entered into the table. Sequence is also somewhat similar to AUTO\_INCREMENT but its has some extra features.

#### Creating Sequence

Syntax to create sequences is,

CREATE **Sequence** *sequence-name*

**start** with *initial-value*

**increment** by *increment-value*

**maxvalue** *maximum-value*

cycle|nocycle

**initial-value** specifies the starting value of the Sequence, **increment-value** is the value by which sequence will be incremented and **maxvalue** specifies the maximum value until which sequence will increment itself.**cycle** specifies that if the maximum value exceeds the set limit, sequence will restart its cycle from the begining. **No cycle** specifies that if sequence exceeds **maxvalue** an error will be thrown.

#### Example to create Sequence

The sequence query is following

CREATE **Sequence** seq\_1

start with 1

increment by 1

maxvalue 999

cycle ;

#### Example to use Sequence

The **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 4 | alex |

The sql query will be,

INSERT into class value(**seq\_1.nextval**,'anu');

Result table will look like,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 4 | Alex |
| 1 | Anu |

Once you use nextval the sequence will increment even if you don't Insert any record into the table.

**6.19 SQL View**

A view in SQL is a logical subset of data from one or more tables. View is used to restrict data access.

Syntax for creating a View,

CREATE or REPLACE **view** *view\_name* AS

SELECT *column\_name*(s)

FROM *table\_name*

WHERE *condition*

**Example of Creating a View**

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 View

CREATE or REPLACE **view** sale\_view as select \* from Sale where customer = 'Alex';

The data fetched from select statement will be stored in another object called **sale\_view**. We can use create seperately and replace too but using both together works better.

**Example of Displaying a View**

Syntax of displaying a view is similar to fetching data from table using Select statement.

SELECT \* from **sale\_view**;

**6.19.1 Force View Creation**

force keyword is used while creating a view. This keyword force to create 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*

**6.19.2 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;

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

**6.19.3 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,

* Simple View
* Complex View

|  |  |
| --- | --- |
| **Simple View** | **Complex View** |
| Created from one table | Created from one or more table |
| Does not contain functions | Contain functions |
| Does not contain groups of data | Contains groups of data |