[**SQL Contents**](http://shettybhavya.blogspot.in/2017/03/sql-contents.html)

|  |  |  |
| --- | --- | --- |
| 1 | Introduction to SQL |  |
|  | a) What is SQL |  |
|  | b)What is Database |  |
|  | c)What is RDBMS |  |
|  | d)DBMS V/S RDBMS |  |
|  | e)SQL Syntax |  |
|  | e)SQL Statement |  |
|  | g)SQL Process |  |
|  | h)SQL Commands |  |
|  | i)SQL Data types |  |
|  | j)SQL Operators  k)SQL  Expressions  l)SQL  Constraints |  |
|  |  |  |
| 2 | SQL Database |  |
|  | 2a)SQL Create Database |  |
|  | 2b)SQL Drop Database |  |
|  | 2c)SQL Rename Database |  |
|  | 2d)SQL Select Database |  |
|  |  |  |
| 3 | SQL Keys |  |
|  | 3a)SQL Primary Key |  |
|  | 3b)SQL Foreign Key |  |
|  | 3c)SQL Composite Key |  |
|  | 3d)SQL Unique Key |  |
|  | 3e)Alternate Key |  |
|  |  |  |
| 4 | Normalization |  |
|  | 4a)First Normal Form (1NF) |  |
|  | 4b)Second Normal  Form(2NF) |  |
|  | 4c)Third Normal Form(3NF) |  |
|  | 4d)Boyce Codd Normal Form(BCNF) |  |
|  |  |  |
| 5 | SQL Table |  |
|  | 5a)SQL Create Table |  |
|  | 5b)SQL Drop Table |  |
|  | 5c)SQL Delete Table |  |
|  | 5d)SQL Rename Table |  |
|  | 5e)SQL Truncate Table |  |
|  | 5f)SQL Copy Table |  |
|  | 5g)SQL Temp Table |  |
|  | 5h)SQL Alter Table |  |
| 6 | SQL Indexes |  |
| 7    8 | SQL  Transactions  SQL Select |  |
|  | 8a)Select Statement |  |
|  | 8b)SQL Select Distinct |  |
|  | 8c)SQL Select Top |  |
|  | 8d)SQL Select AS |  |
|  | 8e)SQL Select IN  8d)SQL Select Into |  |
|  |  |  |
| 9 | SQL Clause |  |
|  | 9a)SQL Where |  |
|  | 9b)SQL AND |  |
|  | 9c)SQL OR  9d)SQL Between  9e)SQL Like  9f)SQL Top, Limit OR ROWNUM  9g)SQL  Having |  |
|  |  |  |
|  |  |  |
| 10 | SQL Order BY |  |
|  | 10a)SQL Order BY Clause |  |
|  | 10b)Order BY ASC |  |
|  | 10c)Order BY DESC |  |
|  | 10d)Order BY Multiple Columns |  |
| 11 | SQL Group BY |  |
| 12 | SQL Insert |  |
|  | 12a)Insert Statement |  |
|  | 12b)Insert Into Values |  |
|  |  |  |
| 13 | SQL Update |  |
|  | 13a)Update Statement |  |
|  | 13b)SQL Update Join |  |
|  | 13c)SQL update date |  |
|  |  |  |
| 14 | SQL Delete |  |
|  | 14a)Delete Statement |  |
|  | 14b)SQL Delete Table |  |
|  | 14c)SQL Delete Row |  |
|  | 14d)SQL Delete All Rows |  |
|  |  |  |
|  |  |  |
| 15 | SQL Join |  |
|  | 15a)SQL Left Join |  |
|  | 15b)SQL Right join |  |
|  | 15c)SQL Full join |  |
|  |  |  |
| 16 | Set Operations in SQL |  |
|  | 16a)Union |  |
|  | 16b)Union ALL |  |
|  | 16c)Intersect |  |
|  | 16d)Minus |  |
| 17 | SQL Auto Increment |  |
| 18 | SQL Views |  |
|  |  |  |
| 19 | Materialized views in Oracle |  |
|  |  |  |
| 20 | SQL Sub queries |  |
| 21 | SQL Injection |  |
| 22 | SQL Aggregate functions |  |
|  | 22a)count() Function |  |
|  | 22b)sum() function |  |
|  | 22c)avg() function |  |
|  | 22d)min() function |  |
|  | 22e)max() function |  |
|  | 22f)first function |  |
|  | 22g)last function |  |
|  |  |  |
| 23 | SQL String Functions |  |
|  |  |  |
| 24 | NVL function |  |
|  | NVL2 Function |  |
|  | Decode Function |  |
|  |  |  |
| 25 | CASE Statement |  |
|  |  |  |
| 26 | SQL Date Functions |  |
|  |  |  |
|  |  |  |

**[SQL INTRO](http://shettybhavya.blogspot.in/2017/03/sql-intro.html)**

What is SQL?

* SQL stands for **Structured Query Language**.
* It is designed for managing data in a relational database management system (RDBMS).
* It is pronounced as S-Q-L or sometime **Sequel**.
* SQL is a database language, it is used for database creation, deletion, fetching rows and modifying rows etc.
* SQL is an ANSI (American National Standards Institute) standard.
* ANSI( is a private [non-profit organization](https://en.wikipedia.org/wiki/Non-profit_organization) that oversees the development of [voluntary consensus standards](https://en.wikipedia.org/wiki/Standardization) for products, services, processes, systems, and personnel in the United States. was originally formed in 1918)

Why SQL?

     SQL is required:

* To create new databases, tables and views
* To insert records in a database
* To update records in a database
* To delete records from a database
* To retrieve data from a database

What SQL does

* With SQL, we can query our database in a numbers of ways.
* With SQL, user can access data from relational database management system.
* It allows user to describe the data.
* It allows user to define the data in database and manipulate it when needed.
* It allows user to create and drop database and table.
* It allows user to create view, stored procedure, function in a database.
* It allows user to set permission on tables, procedure and view.

What is Database

A **database** is *an organized collection of data*.

**Database handlers** create database in such a way that only one set of software program provide access of data to all the users.

The **main purpose** of database is to operate large amount of information by storing, retrieving and managing.

What is RDBMS

**RDBMS** stands for *Relational Database Management Systems.*.

All modern database management systems like SQL, MS SQL Server, IBM DB2, ORACLE, My-SQL and Microsoft Access are based on RDBMS.

How it works

Data is represented in terms of tuples (rows) in RDBMS.

Relational database is most commonly used database. It contains number of tables and each table has its own primary key.

Due to a collection of organized set of tables, data can be accessed easily in RDBMS.

What is table

The RDBMS database uses tables to store data. A table is a collection of related data entries and contains rows and columns to store data.

A table is the simplest example of data storage in RDBMS.

Let's see the example of student table.

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Name** | **AGE** | **COURSE** |
| 1 | sathi | 22 | B.Tech |
| 2 | Aryan | 22 | B.Tech |
| 3 | Mahesh | 22 | BCA |

What is field

Field is a smaller entity of the table which contains specific information about every record in the table. In the above example, the field in the student table consists of id, name, age, and course.

What is row or record

A row of a table is also called record. It contains the specific information of each individual entry in the table. It is a horizontal entity in the table. For example: The above table contains 5 records.

Let's see one record/row in the table.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | Sathi | 22 | B.Tech |

What is column

A column is a vertical entity in the table which contains all information associated with a specific field in a table. For example: "name" is a column in the above table which contains all information about student's name.

|  |
| --- |
| Ajeet |
| Aryan |
| Mahesh |
| Ratan |
| Vimal |

NULL Values

The NULL value of the table specifies that the field has been left blank during record creation. It is totally different from the value filled with zero or a field that contains space.

Data Integrity

There are the following categories of data integrity exist with each RDBMS:

**Entity integrity**: It specifies that there should be no duplicate rows in a table.

**Domain integrity**: It enforces valid entries for a given column by restricting the type, the format, or the range of values.

**Referential integrity**: It specifies that rows cannot be deleted, which are used by other records.

**User-defined integrity**: It enforces some specific business rules that are defined by users. These rules are different from entity, domain or referential integrity.

Difference between DBMS and RDBMS

Although DBMS and RDBMS both are used to store information in physical database but there are some remarkable differences between them.

The main differences between DBMS and RDBMS are given below:

|  |  |  |
| --- | --- | --- |
| **No.** | **DBMS** | **RDBMS** |
| 1) | DBMS applications store **data as file**. | RDBMS applications store **data in a tabular form**. |
| 2) | In DBMS, data is generally stored in either a hierarchical form or a navigational form. | In RDBMS, the tables have an identifier called primary key and the data values are stored in the form of tables. |
| 3) | **Normalization is not** present in DBMS. | **Normalization is** present in RDBMS. |
| 4) | DBMS does **not apply any security** with regards to data manipulation. | RDBMS **defines the integrity constraint** for the purpose of ACID (Atomocity, Consistency, Isolation and Durability) property. |
| 5) | DBMS uses file system to store data, so there will be **no relation betw een the tables**. | in RDBMS, data values are stored in the form of tables, so a**relationship** between these data values will be stored in the form of a table as well. |
| 6) | DBMS has to provide some uniform methods to access the stored information. | RDBMS system supports a tabular structure of the data and a relationship between them to access the stored information. |
| 7) | DBMS **does not support distributed database**. | RDBMS **supports distributed database**. |
| 8) | DBMS is meant to be for small organization and **deal with small data**. it supports **single user**. | RDBMS is designed to **handle large amount of data**. it supports **multiple users**. |
| 9) | Examples of DBMS are file systems, **xml** etc. | Example of RDBMS are **mysql**, **postgre**, **sql server**, **oracle** etc. |

SQL Syntax

SQL follows some unique set of rules and guidelines called syntax. Here, we are providing all the basic SQL syntax.

* SQL is not case sensitive. Generally SQL keywords are written in uppercase.
* SQL statements are dependent on text lines. We can place a single SQL statement on one or multiple text lines.
* You can perform most of the action in a database with SQL statements.

SQL statement

SQL statements are started with any of the SQL commands/keywords like SELECT, INSERT, UPDATE, DELETE, ALTER, DROP etc. and the statement ends with a semicolon (;).

Example of SQL statement:

1.    **SELECT** "column\_name" **FROM** "table\_name";

Why semicolon is used after SQL statements:

Semicolon is used to separate SQL statements. It is a standard way to separate SQL statements in a database system in which more than one SQL statements are used in the same call.

In this tutorial, we will use semicolon at the end of each SQL statement.

SQL Process

When you are executing an SQL command for any RDBMS, the system determines the best way to carry out your request and SQL engine figures out how to interpret the task.

There are various components included in the process. These components are Query Dispatcher, Optimization Engines, Classic Query Engine and SQL Query Engine, etc. Classic query engine handles all non-SQL queries, but SQL query engine won't handle logical files.

Following is a simple diagram showing SQL Architecture:

SQL Commands

These are the some important SQL command:

**SELECT**: it extracts data from a database.

**UPDATE**: it updates data in database.

**DELETE**: it deletes data from database.

**CREATE TABLE**: it creates a new table.

**ALTER TABLE**: it is used to modify the table.

**DROP TABLE**: it deletes a table.

**CREATE DATABASE**: it creates a new database.

**ALTER DATABASE**: It is used to modify a database.

**INSERT INTO**: it inserts new data into a database.

**CREATE INDEX**: it is used to create an index (search key).

**DROP INDEX**: it deletes an index.

The standard SQL commands to interact with relational databases are CREATE, SELECT, INSERT, UPDATE,

DELETE and DROP. These commands can be classified into groups based on their nature:

SQL  COMMAND GROUPS

DDL - Data definition Language:

**Command Description**

CREATE- Creates a new table, a view of a table, or other object in database

ALTER -Modifies an existing database object, such as a table.

DROP -Deletes an entire table, a view of a table or other object in the database.

RENAME-To rename existing table.

DML - Data Manipulation Language:

**Command Description**

INSERT- Creates a record

UPDATE- Modifies records

DELETE -Deletes records

MERGE-To update and insert contents of one table to another table based on condition.

DCL - Data Control Language:

**Command Description**

GRANT- Gives a privilege to user

REVOKE- Takes back privileges granted from user

DQL - Data Query Language:

**Command Description**

SELECT- Retrieves certain records from one or more tables

TCL – Transaction Control Language:

**Command Description**

COMMIT- Save or Enable DML changes to the database.

ROLLBACK-To undo DM changes till in a transaction

SAVEPOINT-To divide a transaction

SQL Data Types

The SQL data type defines a kind of value that a column can contain.

In a database table, every column is required to have a name and a data type.

**Data Type varies from database to database. For example, MySQL supports INT but Oracle supports NUMBER for integer values.**

These are the general data types in SQL.

|  |  |  |
| --- | --- | --- |
| **Data-type** | **Syntax** | **Explanation** |
| Integer | INTEGER | The integer data type is used to specify an integer value. |
| Smallint | SMALLINT | The smallint data type is used to specify small integer value. |
| Numeric | NUMERIC(P,S) | It specifies a numeric value. Here 'p' is precision value and’s’ is scale value. |
| Real | REAL | The real integer is used to specify a single precision floating point number. |
| Decimal | DECIMAL(P,S) | It specifies a decimal value. Here 'p' is precision value and’s’ is scale value. |
| Double precision | DOUBLE PRECISION | It specifies double precision floating point number. |
| Float | FLOAT(P) | It specifies floating-point value e.g. 12.3, 4.5 etc. Here, 'p' is precision value. |
| Character | CHAR(X) | Here, 'x' is the character's number to store. |
| Character varying | VARCHAR2(X) | Here, 'x' is the character's number to store |
| Bit | BIT(X) | Here, 'x' is the number of bits to store |
| Bit varying | BIT VARYING(X) | Here, 'x' is the number of bits to store (length can vary up to x). |
| Date | DATE | It stores year, month and days values. |
| Time | TIME | It stores hour, minute and second values |
| Timestamp | TIMESTAMP | The timestamp data type is used to store year, month, day, hour, minute and second values. |
| Time with time zone | TIME WITH TIME ZONE | It is exactly same as time but also store an offset from UTC of the time specified. |
| Timestamp with time zone | TIMESTAMP with TIME ZONE | It is same as timestamp but also stores an offset from UTC of the time specified. |

SQL Operators

SQL statements generally contain some reserved words or characters that are used to perform operations such as comparison and arithmetical operations etc. These reserved words or characters are known as operators.

Generally there are three types of operators in SQL:

1. SQL Arithmetic Operators
2. SQL Comparison Operators
3. SQL Logical Operators

SQL Arithmetic Operators:

Let's assume two variables "a" and "b". Here "a" is valued 50 and "b" valued 100.

**Example:**

|  |  |  |
| --- | --- | --- |
| **Operators** | **Descriptions** | **Examples** |
| + | It is used to add containing values of both operands | a+b will give 150 |
| - | It subtracts right hand operand from left hand operand | a-b will give -50 |
| \* | It multiply both operands values | a\*b will give 5000 |
| / | It divides left hand operand by right hand operand | b/a will give 2 |
| % | It divides left hand operand by right hand operand and returns reminder | b%a will give 0 |

SQL Comparison Operators:

Let's take two variables "a" and "b" that are valued 50 and 100.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Examine both operands value that are equal or not, if yes condition become true. | (a=b) is not true |
| != | This is used to check the value of both operands equal or not, if not condition becomes true. | (a!=b) is true |
| < > | Examines the operands value equal or not, if values are not equal condition is true | (a<>b) is true |
| > | Examine the left operand value is greater than right Operand, if yes condition becomes true | (a>b) is not true |
| < | Examines the left operand value is less than right Operand, if yes condition becomes True | (a<="" td=""> |
| >= | Examines that the value of left operand is greater than or equal to the value of right operand or not, if yes condition become true | (a>=b) is not true |
| <= | Examines that the value of left operand is less than or equal to the value of right operand or not, if yes condition becomes true | (a<=b) is true |
| !< | Examines that the left operand value is not less than the right operand value | (a!<="" td=""> |
| !> | Examines that the value of left operand is not greater than the value of right operand | (a!>b) is true |

SQL Logical Operators:

This is the list of logical operators used in SQL.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| ALL | This is used to compare a value to all values in another value set. |
| AND | This operator allows the existence of multiple conditions in an SQL statement. |
| ANY | This operator is used to compare the value in list according to the condition. |
| BETWEEN | this operator is used to search for values, that are within a set of values |
| IN | this operator is used to compare a value to that specified list value |
| NOT | the NOT operator reverse the meaning of any logical operator |
| OR | this operator is used to combine multiple conditions in SQL statements |
| EXISTS | the EXISTS operator is used to search for the presence of a row in a specified  table |
| LIKE | this operator is used to compare a value to similar values using wildcard operator |

SQL – Expressions

An expression is a combination of one or more values, operators, and SQL functions that evaluate to a value.

SQL EXPRESSIONs are like formulas and they are written in query language. You can also use them to query the database for specific set of data.

Syntax:

Consider the basic syntax of the SELECT statement as follows:

SELECT column1, column2, columnN

FROM table\_name

WHERE [CONDITION|EXPRESSION];

There are different types of SQL expressions, which are mentioned below:

SQL - Boolean Expressions:

SQL Boolean Expressions fetch the data on the basis of matching single value. Following is the syntax:

SELECT column1, column2, columnN

FROM table\_name

WHERE SINGLE VALUE MATCHING EXPRESSION;

Consider the CUSTOMERS table having the following records:

SQL> SELECT \* FROM CUSTOMERS;

+----+----------+-----+-----------+----------+

| ID | NAME     | AGE | ADDRESS   | SALARY   |

+----+----------+-----+-----------+----------+

|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |

|  2 | Khilan   |  25 | Delhi     |  1500.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

|  4 | Chaitali |  25 | Mumbai    |  6500.00 |

|  5 | Hardik   |  27 | Bhopal    |  8500.00 |

|  6 | Komal    |  22 | MP        |  4500.00 |

|  7 | Muffy    |  24 | Indore    | 10000.00 |

+----+----------+-----+-----------+----------+

7 rows in set (0.00 sec)

Here is simple example showing usage of SQL Boolean Expressions:

SQL> SELECT \* FROM CUSTOMERS WHERE SALARY = 10000;

+----+-------+-----+---------+----------+

| ID | NAME  | AGE | ADDRESS | SALARY   |

+----+-------+-----+---------+----------+

|  7 | Muffy |  24 | Indore  | 10000.00 |

+----+-------+-----+---------+----------+

1 row in set (0.00 sec)

SQL - Numeric Expression:

This expression is used to perform any mathematical operation in any query. Following is the syntax:

SELECT numerical\_expression as  OPERATION\_NAME

[FROM table\_name

WHERE CONDITION] ;

Here numerical\_expression is used for mathematical expression or any formula. Following is a simple examples showing usage of SQL Numeric Expressions:

SQL> SELECT (15 + 6) AS ADDITION

+----------+

| ADDITION |

+----------+

|       21 |

+----------+

1 row in set (0.00 sec)

There are several built-in functions like avg(), sum(), count(), etc., to perform what is known as aggregate data calculations against a table or a specific table column.

SQL> SELECT COUNT(\*) AS "RECORDS" FROM CUSTOMERS;

+---------+

| RECORDS |

+---------+

|       7 |

+---------+

1 row in set (0.00 sec)

SQL - Date Expressions:

Date Expressions return current system date and time values:

SQL>  SELECT CURRENT\_TIMESTAMP;

+---------------------+

| Current\_Timestamp   |

+---------------------+

| 2009-11-12 06:40:23 |

+---------------------+

1 row in set (0.00 sec)

Another date expression is as follows:

SQL>  SELECT  GETDATE();;

+-------------------------+

| GETDATE                 |

+-------------------------+

| 2009-10-22 12:07:18.140 |

+-------------------------+

**Constraints**

### SQL – Constraints

Constraints are the rules enforced on data columns on table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.

Constraints could be column level or table level. Column level constraints are applied only to one column, whereas table level constraints are applied to the whole table.

Following are commonly used constraints available in SQL. These constraints have already been discussed in [SQL - RDBMS Concepts](https://www.tutorialspoint.com/sql/sql-rdbms-concepts.htm) chapter but its worth to revise them at this point.

        [NOT NULL Constraint](https://www.tutorialspoint.com/sql/sql-not-null.htm): Ensures that a column cannot have NULL value.

CREATE TABLE Persons (

    ID int NOT NULL,

    LastName varchar(255) NOT NULL,

    FirstName varchar(255) NOT NULL,

    Age int

);

        [DEFAULT Constraint](https://www.tutorialspoint.com/sql/sql-default.htm): Provides a default value for a column when none is specified.

CREATE TABLE Persons (

    ID int NOT NULL,

    LastName varchar(255) NOT NULL,

    FirstName varchar(255),

    Age int,

    City varchar(255) DEFAULT 'Sandnes'

);

The DEFAULT constraint can also be used to insert system values, by using functions like GETDATE():

CREATE TABLE Orders (

    ID int NOT NULL,

    OrderNumber int NOT NULL,

    OrderDate date DEFAULT GETDATE()

);

        [UNIQUE Constraint](https://www.tutorialspoint.com/sql/sql-unique.htm): Ensures that all values in a column are different.

        [PRIMARY Key](https://www.tutorialspoint.com/sql/sql-primary-key.htm): Uniquely identified each rows/records in a database table.

        [FOREIGN Key](https://www.tutorialspoint.com/sql/sql-foreign-key.htm): Uniquely identified a rows/records in any another database table.

        [CHECK Constraint](https://www.tutorialspoint.com/sql/sql-check.htm): The CHECK constraint ensures that all values in a column satisfy certain conditions.

CREATE TABLE Persons (

ID int NOT NULL,

LastName varchar(255) NOT NULL,

      FirstName varchar(255),

Age int,

   CHECK (Age>=18));

        [INDEX](https://www.tutorialspoint.com/sql/sql-index.htm): Use to create and retrieve data from the database very quickly.

Constraints can be specified when a table is created with the CREATE TABLE statement or you can use ALTER TABLE statement to create constraints even after the table is created.

## 

## 

## Dropping Constraints:

Any constraint that you have defined can be dropped using the ALTER TABLE command with the DROP CONSTRAINT option.

For example, to drop the primary key constraint in the EMPLOYEES table, you can use the following command:

ALTER TABLE EMPLOYEES DROP CONSTRAINT EMPLOYEES\_PK;

Some implementations may provide shortcuts for dropping certain constraints. For example, to drop the primary key constraint for a table in Oracle, you can use the following command:

ALTER TABLE EMPLOYEES DROP PRIMARY KEY;

Some implementations allow you to disable constraints. Instead of permanently dropping a constraint from the database, you may want to temporarily disable the constraint and then enable it later.

## Integrity Constraints:

Integrity constraints are used to ensure accuracy and consistency of data in a relational database. Data integrity is handled in a relational database through the concept of referential integrity.

There are many types of integrity constraints that play a role in referential integrity (RI). These constraints include Primary Key, Foreign Key, Unique Constraints and other constraints mentioned above.

CREATE Database

SQL CREATE Database

The **SQL CREATE DATABASE** statement is used by a developer to create a database.

Let's see the syntax of SQL CREATE DATABASE:

**CREATE** **DATABASE** database\_name;

If you want to add tables in that database, you can use CREATE TABLE statement.

Create Database in MySQL

In MySQL, same command is used to create a database.

**CREATE** **DATABASE** database\_name;

Create Database in Oracle

You don't need to create database in Oracle. In Oracle database, you can create tables directly.

You can also rename, drop and select database that is covered in next pages.

**We should always remember that database name should be unique in the RDBMS.**

SQL DROP Database

**SQL DROP statement is used to delete or remove indexes from a table in the database.**

If you want to delete or drop an existing database in a SQL schema, you can use SQL DROP DATABASE

Let's see the syntax of SQL DROP DATABASE:

1.    **DROP** **DATABASE** database\_name;

**If you delete or drop the database, all the tables and views will also be deleted. So be careful while using this command.**

SQL RENAME Database

SQL RENAME DATABASE is used when you need to change the name of your database. Sometimes it is used because you think that the original name is not more relevant to the database or you want to give a temporary name to that database.

Let's see how to rename Mysql and SQL Server databases.

Rename MySQL database

To rename the mysql database, you need to follow the following syntax:

1.    RENAME **DATABASE** old\_db\_name **TO** new\_db\_name

SQL SELECT Database

In MySQL database, you need to select a database first before executing any query on table, view etc. To do so, we use following query:

1.    USE **DATABASE** database\_name;

In oracle, you don't need to select database.

**SQL KEYS**

SQL PRIMARY KEY

A column or columns is called **primary key (PK)** that *uniquely identifies each row in the table*.

If you want to create a primary key, you should define a PRIMARY KEY constraint when you create or modify a table.

When multiple columns are used as a primary key, it is known as **composite primary key**.

Points to remember for primary key:

* Primary key enforces the entity integrity of the table.
* Primary key always has unique data.
* A primary key length cannot be exceeded than 900 bytes.
* A primary key cannot have null value.
* There can be no duplicate value for a primary key.
* A table can contain only one primary key constraint.

**When we specify a primary key constraint for a table, database engine automatically creates a**

The main ad unique index for the primary key column.

         In short, Primary key is by default NOT NULL, INDEXED, and  UNIQUE..!

Main advantage of primary key:

The main advantage of this uniqueness is that we get **fast access**.

In oracle, it is not allowed for a primary key to contain more than 32 columns.

SQL primary key for one column:

The following SQL command creates a PRIMARY KEY on the "S\_Id" column when the "students" table is created.

**MySQL:**

1.    **CREATE** **TABLE** students

2.    (

3.    S\_Id **int** NOT NULL,

4.    LastName **varchar** (255) NOT NULL,

5.    FirstName **varchar** (255),

6.    Address **varchar** (255),

7.    City **varchar** (255),

8.    **PRIMARY** **KEY** (S\_Id)

9.    )

SQL primary key for multiple columns:

**MySQL, SQL Server, Oracle, MS Access:**

1.    **CREATE** **TABLE** students

2.    (

3.    S\_Id **int** NOT NULL,

4.    LastName **varchar** (255) NOT NULL,

5.    FirstName **varchar** (255),

6.    Address **varchar** (255),

7.    City **varchar** (255),

8.    **CONSTRAINT** pk\_StudentID **PRIMARY** **KEY** (S\_Id, LastName)

9.    )

**Note:**you should note that in the above example there is only one PRIMARY KEY (pk\_StudentID). However it is made up of two columns (S\_Id and LastName).

SQL primary key on ALTER TABLE

When table is already created and you want to create a PRIMARY KEY constraint on the ?S\_Id? column you should use the following SQL:

**Primary key on one column:**

1.    **ALTER** **TABLE** students

2.    **ADD** **PRIMARY** **KEY** (S\_Id)

**Primary key on multiple column:**

1.    **ALTER** **TABLE** students

2.    **ADD** **CONSTRAINT** pk\_StudentID **PRIMARY** **KEY** (S\_Id,LastName)

When you use ALTER TABLE statement to add a primary key, the primary key columns must not contain NULL values (when the table was first created).

How to DROP a PRIMARY KEY constraint?

If you want to DROP (remove) a primary key constraint, you should use following syntax:

**MySQL:**

1.    **ALTER** **TABLE** students

2.    **DROP** **PRIMARY** **KEY**

SQL FOREIGN KEY

In the relational databases, a foreign key is a field or a column that is used to establish a link between two tables.

In simple words you can say that, a foreign key in one table used to point primary key in another table.

Let us take an example to explain it:

Here are two tables first one is students table and second is orders table.

Here orders are given by students.

**First table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **S\_Id** | **LastName** | **FirstName** | **CITY** |
| 1 | MAURYA | AJEET | ALLAHABAD |
| 2 | JAISWAL | RATAN | GHAZIABAD |
| 3 | ARORA | SAUMYA | MODINAGAR |

**Second table:**

|  |  |  |
| --- | --- | --- |
| **O\_Id** | **OrderNo** | **S\_Id** |
| 1 | 99586465 | 2 |
| 2 | 78466588 | 2 |
| 3 | 22354846 | 3 |
| 4 | 57698656 | 1 |

Here you see that "S\_Id" column in the "Orders" table points to the "S\_Id" column in "Students" table.

* The "S\_Id" column in the "Students" table is the PRIMARY KEY in the "Students" table.
* The "S\_Id" column in the "Orders" table is a FOREIGN KEY in the "Orders" table.

The foreign key constraint is generally prevents action that destroy links between tables.

It also prevents invalid data to enter in foreign key column.

SQL FOREIGN KEY constraint ON CREATE TABLE:

(Defining a foreign key constraint on single column)

To create a foreign key on the "S\_Id" column when the "Orders" table is created:

**MySQL:**

1.    **CREATE** **TABLE** orders

2.    (

3.    O\_Id **int** NOT NULL,

4.    Order\_No  **int** NOT NULL,

5.    S\_Id **int**,

6.    PRIMAY **KEY** (O\_Id),

7.    **FOREIGN** **KEY** (S\_Id) **REFERENCES** Persons (S\_Id)

8.    )

SQL FOREIGN KEY constraint for ALTER TABLE:

If the Order table is already created and you want to create a FOREIGN KEY constraint on the ?S\_Id? column, you should write the following syntax:

**Defining a foreign key constraint on single column:**

**MySQL / SQL Server / Oracle / MS Access:**

1.    **ALTER** **TABLE** Orders

2.    **ADD** **CONSTRAINT** fk\_PerOrders

3.    **FOREIGN** **KEY**(S\_Id)

4.    **REFERENCES** Students (S\_Id)

DROP SYNTAX for FOREIGN KEY COSTRAINT:

If you want to drop a FOREIGN KEY constraint, use the following syntax:

**MySQL:**

1.    **ALTER** **TABLE** Orders

2.    ROP **FOREIGN** **KEY** fk\_PerOrders

Difference between primary key and foreign key in SQL:

These are some important difference between primary key and foreign key in SQL-

Primary key cannot be null on the other hand foreign key can be null.

Primary key is always unique while foreign key can be duplicated.

Primary key uniquely identify a record in a table while foreign key is a field in a table that is primary key in another table.

There is only one primary key in the table on the other hand we can have more than one foreign key in the table.

By default primary key adds a clustered index on the other hand foreign key does not automatically create an index, clustered or non-clustered. You must manually create an index for foreign key.

SQL Composite Key

A composite key is a combination of two or more columns in a table that can be used to uniquely identify each row in the table when the columns are combined uniqueness is guaranteed, but when it taken individually it does not guarantee uniqueness.

In other words we can say that:

Composite key is a key which is the combination of more than one field or column of a given table. It may be a candidate key or primary key.

Columns that make up the composite key can be of different data types.

**SQL Syntax to specify composite key:**

1.    **CREATE** **TABLE** TABLE\_NAME

2.    (COLUMN\_1, DATA\_TYPE\_1,

3.    COLUMN\_2, DATA\_TYPE\_2,

4.    ???

5.    **PRIMARY** **KEY** (COLUMN\_1, COLUMN\_2, ...));

In all cases composite key created consist of COLUMN1 and COLUMN2.

MySQL:

1.    **CREATE** **TABLE** SAMPLE\_TABLE

2.    (COL1 **integer**,

3.    COL2 **varchar**(30),

4.    COL3 **varchar**(50),

5.    **PRIMARY** **KEY** (COL1, COL2));

Unique Key in SQL

A unique key is a set of one or more than one fields/columns of a table that uniquely identify a record in a database table.

You can say that it is little like primary key but it can accept only one null value and it cannot have duplicate values.

The unique key and primary key both provide a guarantee for uniqueness for a column or a set of columns.

There is an automatically defined unique key constraint within a primary key constraint.

There may be many unique key constraints for one table, but only one PRIMARY KEY constraint for one table.

Alternate Key in SQL

Alternate key is a secondary key it can be simple to understand by an example:

Let's take an example of student it can contain NAME, ROLL NO., ID and CLASS.

Here ROLL NO. is primary key and rest of all columns like NAME, ID and CLASS are alternate keys.

If a table has more than one candidate key, one of them will become the primary key and rest of all are called alternate keys.

An alternate key is just a candidate key that has not been selected as the primary key.

**Normalization of Database**

Database Normalisation is a technique of organizing the data in the database. Normalization is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anomalies. It is a multi-step process that puts data into tabular form by removing duplicated data from the relation tables.

Normalization is used for mainly two purpose,

         Eliminating reduntant(useless) data.

         Ensuring data dependencies make sense i.e data is logically stored.

**Problem Without Normalization**

Without Normalization, it becomes difficult to handle and update the database, without facing data loss. Insertion, Updation and Deletion Anamolies are very frequent if Database is not Normalized. To understand these anomalies let us take an example of **Student** table.

|  |  |  |  |
| --- | --- | --- | --- |
| **S\_id** | **S\_Name** | **S\_Address** | **Subject\_opted** |
| 401 | Adam | Noida | Bio |
| 402 | Alex | Panipat | Maths |
| 403 | Stuart | Jammu | Maths |
| 404 | Adam | Noida | Physics |

         **Updation Anamoly:** To update address of a student who occurs twice or more than twice in a table, we will have to update **S\_Address** column in all the rows, else data will become inconsistent.

         **Insertion Anamoly:** Suppose for a new admission, we have a Student id (S\_id), name and address of a student but if student has not opted for any subjects yet then we have to insert **NULL** there, leading to Insertion Anamoly.

         **Deletion Anamoly:** If (S\_id) 401 has only one subject and temporarily he drops it, when we delete that row, entire student record will be deleted along with it.

**Normalization Rule**

Normalization rule are divided into following normal form.

1.    First Normal Form

2.    Second Normal Form

3.    Third Normal Form

4.    BCNF

**First Normal Form (1NF)**

As per First Normal Form, no two Rows of data must contain repeating group of information i.e each set of column must have a unique value, such that multiple columns cannot be used to fetch the same row. Each table should be organized into rows, and each row should have a primary key that distinguishes it as unique.

The **Primary key** is usually a single column, but sometimes more than one column can be combined to create a single primary key. For example consider a table which is not in First normal form

**Student Table :**

|  |  |  |
| --- | --- | --- |
| **Student** | **Age** | **Subject** |
| Adam | 15 | Biology, Maths |
| Alex | 14 | Maths |
| Stuart | 17 | Maths |

In First Normal Form, any row must not have a column in which more than one value is saved, like separated with commas. Rather than that, we must separate such data into multiple rows.

**Student Table following 1NF will be :**

|  |  |  |
| --- | --- | --- |
| **Student** | **Age** | **Subject** |
| Adam | 15 | Biology |
| Adam | 15 | Maths |
| Alex | 14 | Maths |
| Stuart | 17 | Maths |

Using the First Normal Form, data redundancy increases, as there will be many columns with same data in multiple rows but each row as a whole will be unique.

**Second Normal Form (2NF)**

As per the Second Normal Form there must not be any partial dependency of any column on primary key. It means that for a table that has concatenated primary key, each column in the table that is not part of the primary key must depend upon the entire concatenated key for its existence. If any column depends only on one part of the concatenated key, then the table fails **Second normal form**.

In example of First Normal Form there are two rows for Adam, to include multiple subjects that he has opted for. While this is searchable, and follows First normal form, it is an inefficient use of space. Also in the above Table in First Normal Form, while the candidate key is {**Student**, **Subject**}, **Age** of Student only depends on Student column, which is incorrect as per Second Normal Form. To achieve second normal form, it would be helpful to split out the subjects into an independent table, and match them up using the student names as foreign keys.

**New Student Table following 2NF will be :**

|  |  |
| --- | --- |
| **Student** | **Age** |
| Adam | 15 |
| Alex | 14 |
| Stuart | 17 |

In Student Table the candidate key will be **Student** column, because all other column i.e **Age** is dependent on it.

**New Subject Table introduced for 2NF will be :**

|  |  |
| --- | --- |
| **Student** | **Subject** |
| Adam | Biology |
| Adam | Maths |
| Alex | Maths |
| Stuart | Maths |

In Subject Table the candidate key will be {**Student**, **Subject**} column. Now, both the above tables qualifies for Second Normal Form and will never suffer from Update Anomalies. Although there are a few complex cases in which table in Second Normal Form suffers Update Anomalies, and to handle those scenarios Third Normal Form is there.

**Third Normal Form (3NF)**

**Third Normal form** applies that every non-prime attribute of table must be dependent on primary key, or we can say that, there should not be the case that a non-prime attribute is determined by another non-prime attribute. So this *transitive functional dependency* should be removed from the table and also the table must be in **Second Normal form**. For example, consider a table with following fields.

**Student\_Detail Table :**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Student\_id** | **Student\_name** | **DOB** | **Street** | **city** | **State** | **Zip** |

In this table Student\_id is Primary key, but street, city and state depends upon Zip. The dependency between zip and other fields is called **transitive dependency**. Hence to apply **3NF**, we need to move the street, city and state to new table, with **Zip** as primary key.

**New Student\_Detail Table :**

|  |  |  |  |
| --- | --- | --- | --- |
| **Student\_id** | **Student\_name** | **DOB** | **Zip** |

**Address Table :**

|  |  |  |  |
| --- | --- | --- | --- |
| **Zip** | **Street** | **City** | **state** |

The advantage of removing transitive dependency is,

         Amount of data duplication is reduced.

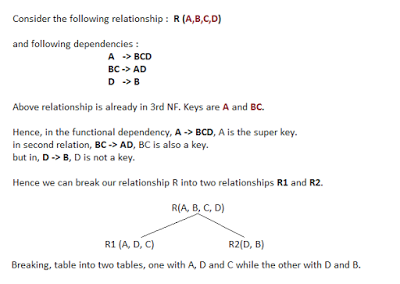
         Data integrity achieved.

**Boyce and Codd Normal Form (BCNF)**

**Boyce and Codd Normal Form** is a higher version of the Third Normal form. This form deals with certain type of anamoly that is not handled by 3NF. A 3NF table which does not have multiple overlapping candidate keys is said to be in BCNF. For a table to be in BCNF, following conditions must be satisfied:

         R must be in 3rd Normal Form

         and, for each functional dependency ( X -> Y ), X should be a super Key.

[](https://4.bp.blogspot.com/-Q_5kR59Vx7I/WMoGEvvY1CI/AAAAAAAACzA/hGw8P8-39l4QztZKWJ3YjwZTh-AOOrAdACLcB/s1600/Untitled.png)

**SQL Table**

SQL CREATE TABLE

SQL CREATE TABLE statement is used to create table in a database.

If you want to create a table, you should name the table and define its column and each column's data type.

Let's see the simple syntax to create the table.

1.    **create** **table** "tablename"

2.    ("column1" "data type",

3.    "column2" "data type",

4.    "column3" "data type",

5.    ...

6.    "columnN" "data type");

The data type of the columns may vary from one database to another. For example, NUMBER is supported in Oracle database for integer value whereas INT is supported in MySQL.

SQL CREATE TABLE Example in MySQL

1.    **CREATE** **TABLE** Employee

2.    (

3.    EmployeeID **int**,

4.    FirstName **varchar**(255),

5.    LastName **varchar**(255),

6.    Email **varchar**(255),

7.    AddressLine **varchar**(255),

8.    City **varchar**(255)

9.    );

SQL DROP TABLE

A SQL DROP TABLE statement is used to delete a table definition and all data from a table.

Let's see the syntax to drop the table from the database.

**DROP** **TABLE** "table\_name";

SQL DELETE TABLE

The DELETE statement is used to delete rows from a table. If you want to remove a specific row from a table you should use WHERE condition.

**DELETE** **FROM** table\_name [**WHERE** condition];

But if you do not specify the WHERE condition it will remove all the rows from the table.

**DELETE** **FROM** table\_name;

Difference between DELETE and TRUNCATE statements

There is a slight difference b/w delete and truncate statement. The **DELETE statement** only deletes the rows from the table based on the condition defined by WHERE clause or delete all the rows from the table when condition is not specified.

But it does not free the space containing by the table.

The **TRUNCATE statement:** it is used to delete all the rows from the table **and free the containing space.**

Execute the following query to truncate the table:

**TRUNCATE** **TABLE** employee;

Difference b/w DROP and TRUNCATE statements

When you use the drop statement it deletes the table's row together with the table's definition so all the relationships of that table with other tables will no longer be valid.

**When you drop a table:**

* Table structure will be dropped
* Relationship will be dropped
* Integrity constraints will be dropped
* Access privileges will also be dropped

On the other hand when we **TRUNCATE** a table, the table structure remains the same, so you will not face any of the above problems.

SQL RENAME TABLE

**SQL RENAME TABLE** syntax is used to change the name of a table. Sometimes, we choose non-meaningful name for the table. So it is required to be changed.

Let's see the syntax to rename a table from the database.

1.    **ALTER** **TABLE** table\_name

2.    RENAME **TO** new\_table\_name;

SQL TRUNCATE TABLE

A truncate SQL statement is used to remove all rows (complete data) from a table. It is similar to the DELETE statement with no WHERE clause.

TRUNCATE TABLE Vs DELETE TABLE

Truncate table is faster and uses lesser resources than DELETE TABLE command.

TRUNCATE TABLE Vs DROP TABLE

Drop table command can also be used to delete complete table but it deletes table structure too. TRUNCATE TABLE doesn't delete the structure of the table.

Let's see the syntax to truncate the table from the database.

1.    **TRUNCATE** **TABLE** table\_name;

For example, you can write following command to truncate the data of employee table

1.    **TRUNCATE** **TABLE** Employee;

**Note:** The rollback process is not possible after truncate table statement. Once you truncate a table you cannot use a flashback table statement to retrieve the content of the table.

SQL COPY TABLE

If you want to copy a SQL table into another table in the same SQL server database, it is possible by using the select statement.

The syntax of copying table from one to another is given below:

1.    **SELECT** \* **INTO** <destination\_table> **FROM** <source\_table>

For example, you can write following command to copy the records of hr\_employee table into employee table.

1.    **SELECT** \* **INTO** admin\_employee **FROM** hr\_employee;

**Note: SELECT INTO is totally different from INSERT INTO statement.**

SQL TEMP TABLE

The concept of temporary table is introduced by SQL server. It helps developers in many ways:

**Temporary tables** can be created at run-time and can do all kinds of operations that a normal table can do. These temporary tables are created inside tempdb database.

There are two types of temp tables based on the behavior and scope.

1. Local Temp Variable
2. Global Temp Variable

Local Temp Variable

Local temp tables are only available at current connection time. It is automatically deleted when user disconnects from instances. It is started with hash (#) sign.

1.    **CREATE** **TABLE** #**local** **temp** **table** (

2.    User id **int**,

3.    Username **varchar** (50),

4.    User address **varchar** (150)

5.    )

Global Temp Variable

Global temp tables name starts with double hash (##). Once this table is created, it is like a permanent table. It is always ready for all users and not deleted until the total connection is withdrawn.

1.    **CREATE** **TABLE** ##new **global** **temp** **table** (

2.    User id **int**,

3.    User **name** **varchar** (50),

4.    User address **varchar** (150)

5.    )

SQL ALTER TABLE

The ALTER TABLE statement is used to add, modify or delete columns in an existing table. It is also used to rename a table.

You can also use SQL ALTER TABLE command to add and drop various constraints on an existing table.

SQL ALTER TABLE Add Column

If you want to add columns in SQL table, the SQL alter table syntax is given below:

1.    **ALTER** **TABLE** table\_name **ADD** column\_name **column**-definition;

If you want to add multiple columns in table, the SQL table will be

1.    **ALTER** **TABLE** table\_name

2.    **ADD** (column\_1 **column**-definition,

3.               column\_2 **column**-definition,

4.               .....

5.               column\_n **column**-definition);

SQL ALTER TABLE Modify Column

If you want to modify an existing column in SQL table, syntax is given below:

1.    **ALTER** **TABLE** table\_name **MODIFY** column\_name column\_type;

If you want to modify multiple columns in table, the SQL table will be

1.    **ALTER** **TABLE** table\_name

2.    **MODIFY** (column\_1 column\_type,

3.                      column\_2 column\_type,

4.                     .....

5.                      column\_n column\_type);

SQL ALTER TABLE DROP Column

The syntax of alter table drop column is given below:

1.    **ALTER** **TABLE** table\_name **DROP** **COLUMN** column\_name;

SQL ALTER TABLE RENAME Column

The syntax of alter table rename column is given below:

1.    **ALTER** **TABLE** table\_name

2.    RENAME **COLUMN** old\_name **to** new\_name

### SQL – Indexes & Transactions

Indexes are special lookup tables that the database search engine can use to speed up data retrieval. Simply put, an index is a pointer to data in a table. An index in a database is very similar to an index in the back of a book.

For example, if you want to reference all pages in a book that discuss a certain topic, you first refer to the index, which lists all topics alphabetically and are then referred to one or more specific page numbers.

An index helps speed up SELECT queries and WHERE clauses, but it slows down data input, with UPDATE and INSERT statements. Indexes can be created or dropped with no effect on the data.

Creating an index involves the CREATE INDEX statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in ascending or descending order.

Indexes can also be unique, similar to the UNIQUE constraint, in that the index prevents duplicate entries in the column or combination of columns on which there's an index.

## The CREATE INDEX Command:

The basic syntax of **CREATE INDEX** is as follows:

CREATE INDEX index\_name ON table\_name;

## Single-Column Indexes:

A single-column index is one that is created based on only one table column. The basic syntax is as follows:

CREATE INDEX index\_name

ON table\_name (column\_name);

## Unique Indexes:

Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows:

CREATE UNIQUE INDEX index\_name

on table\_name (column\_name);

## 

## 

## Composite Indexes:

A composite index is an index on two or more columns of a table. The basic syntax is as follows:

CREATE INDEX index\_name

on table\_name (column1, column2);

Whether to create a single-column index or a composite index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions.

Should there be only one column used, a single-column index should be the choice. Should there be two or more columns that are frequently used in the WHERE clause as filters, the composite index would be the best choice.

## Implicit Indexes:

Implicit indexes are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

## The DROP INDEX Command:

An index can be dropped using SQL **DROP** command. Care should be taken when dropping an index because performance may be slowed or improved.

The basic syntax is as follows:

DROP INDEX index\_name;

You can check [INDEX Constraint](https://www.tutorialspoint.com/sql/sql-index.htm) chapter to see actual examples on Indexes.

## When should indexes be avoided?

Although indexes are intended to enhance a database's performance, there are times when they should be avoided. The following guidelines indicate when the use of an index should be reconsidered:

        Indexes should not be used on small tables.

        Tables that have frequent, large batch update or insert operations.

        Indexes should not be used on columns that contain a high number of NULL values.

        Columns that are frequently manipulated should not be indexed.

# SQL – Transactions

A transaction is a unit of work that is performed against a database. Transactions are units or sequences of work accomplished in a logical order, whether in a manual fashion by a user or automatically by some sort of a database program

Properties of Transactions:

Transactions have the following four standard properties, usually referred to by the acronym ACID:

        **Atomicity:** ensures that all operations within the work unit are completed successfully; otherwise, the transaction is aborted at the point of failure, and previous operations are rolled back to their former state.

        **Consistency:** ensures that the database properly changes states upon a successfully committed transaction.

        **Isolation:** enables transactions to operate independently of and transparent to each other.

        **Durability:** ensures that the result or effect of a committed transaction persists in case of a system failure.

Transaction Control:

There are following commands used to control transactions:

        **COMMIT:** to save the changes.

        **ROLLBACK:** to rollback the changes.

        **SAVEPOINT:** creates points within groups of transactions in which to ROLLBACK

        **SET TRANSACTION:** Places a name on a transaction.

Transactional control commands are only used with the DML commands INSERT, UPDATE and DELETE only. They can not be used while creating tables or dropping them because these operations are automatically commited in the database.

The COMMIT Command:

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.

The COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command.

The syntax for COMMIT command is as follows:

COMMIT;

Example:

Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| ID | NAME     | AGE | ADDRESS   | SALARY   |

+----+----------+-----+-----------+----------+

|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |

|  2 | Khilan   |  25 | Delhi     |  1500.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

|  4 | Chaitali |  25 | Mumbai    |  6500.00 |

|  5 | Hardik   |  27 | Bhopal    |  8500.00 |

|  6 | Komal    |  22 | MP        |  4500.00 |

|  7 | Muffy    |  24 | Indore    | 10000.00 |

+----+----------+-----+-----------+----------+

Following is the example which would delete records from the table having age = 25 and then COMMIT the changes in the database.

SQL> DELETE FROM CUSTOMERS

     WHERE AGE = 25;

SQL> COMMIT;

As a result, two rows from the table would be deleted and SELECT statement would produce the following result:

+----+----------+-----+-----------+----------+

| ID | NAME     | AGE | ADDRESS   | SALARY   |

+----+----------+-----+-----------+----------+

|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

|  5 | Hardik   |  27 | Bhopal    |  8500.00 |

|  6 | Komal    |  22 | MP        |  4500.00 |

|  7 | Muffy    |  24 | Indore    | 10000.00 |

+----+----------+-----+-----------+----------+

The ROLLBACK Command:

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database.

The ROLLBACK command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

The syntax for ROLLBACK command is as follows:

ROLLBACK;

Example:

Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| ID | NAME     | AGE | ADDRESS   | SALARY   |

+----+----------+-----+-----------+----------+

|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |

|  2 | Khilan   |  25 | Delhi     |  1500.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

|  4 | Chaitali |  25 | Mumbai    |  6500.00 |

|  5 | Hardik   |  27 | Bhopal    |  8500.00 |

|  6 | Komal    |  22 | MP        |  4500.00 |

|  7 | Muffy    |  24 | Indore    | 10000.00 |

+----+----------+-----+-----------+----------+

Following is the example, which would delete records from the table having age = 25 and then ROLLBACK the changes in the database.

SQL> DELETE FROM CUSTOMERS

     WHERE AGE = 25;

SQL> ROLLBACK;

As a result, delete operation would not impact the table and SELECT statement would produce the following result:

+----+----------+-----+-----------+----------+

| ID | NAME     | AGE | ADDRESS   | SALARY   |

+----+----------+-----+-----------+----------+

|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |

|  2 | Khilan   |  25 | Delhi     |  1500.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

|  4 | Chaitali |  25 | Mumbai    |  6500.00 |

|  5 | Hardik   |  27 | Bhopal    |  8500.00 |

|  6 | Komal    |  22 | MP        |  4500.00 |

|  7 | Muffy    |  24 | Indore    | 10000.00 |

The SAVEPOINT Command:

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

The syntax for SAVEPOINT command is as follows:

SAVEPOINT SAVEPOINT\_NAME;

This command serves only in the creation of a SAVEPOINT among transactional statements. The ROLLBACK command is used to undo a group of transactions.

The syntax for rolling back to a SAVEPOINT is as follows:

ROLLBACK TO SAVEPOINT\_NAME;

Following is an example where you plan to delete the three different records from the CUSTOMERS table. You want to create a SAVEPOINT before each delete, so that you can ROLLBACK to any SAVEPOINT at any time to return the appropriate data to its original state:

Example:

Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| ID | NAME     | AGE | ADDRESS   | SALARY   |

+----+----------+-----+-----------+----------+

|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |

|  2 | Khilan   |  25 | Delhi     |  1500.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

|  4 | Chaitali |  25 | Mumbai    |  6500.00 |

|  5 | Hardik   |  27 | Bhopal    |  8500.00 |

|  6 | Komal    |  22 | MP        |  4500.00 |

|  7 | Muffy    |  24 | Indore    | 10000.00 |

+----+----------+-----+-----------+----------+

Now, here is the series of operations:

SQL> SAVEPOINT SP1;

Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=1;

1 row deleted.

SQL> SAVEPOINT SP2;

Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=2;

1 row deleted.

SQL> SAVEPOINT SP3;

Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=3;

1 row deleted.

Now that the three deletions have taken place, say you have changed your mind and decided to ROLLBACK to the SAVEPOINT that you identified as SP2. Because SP2 was created after the first deletion, the last two deletions are undone:

SQL> ROLLBACK TO SP2;

Rollback complete.

Notice that only the first deletion took place since you rolled back to SP2:

SQL> SELECT \* FROM CUSTOMERS;

+----+----------+-----+-----------+----------+

| ID | NAME     | AGE | ADDRESS   | SALARY   |

+----+----------+-----+-----------+----------+

|  2 | Khilan   |  25 | Delhi     |  1500.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

|  4 | Chaitali |  25 | Mumbai    |  6500.00 |

|  5 | Hardik   |  27 | Bhopal    |  8500.00 |

|  6 | Komal    |  22 | MP        |  4500.00 |

|  7 | Muffy    |  24 | Indore    | 10000.00 |

+----+----------+-----+-----------+----------+

6 rows selected.

The RELEASE SAVEPOINT Command:

The RELEASE SAVEPOINT command is used to remove a SAVEPOINT that you have created.

The syntax for RELEASE SAVEPOINT is as follows:

RELEASE SAVEPOINT SAVEPOINT\_NAME;

Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the SAVEPOINT.

The SET TRANSACTION Command:

The SET TRANSACTION command can be used to initiate a database transaction. This command is used to specify characteristics for the transaction that follows.

For example, you can specify a transaction to be read only, or read write.

The syntax for SET TRANSACTION is as follows:

SET TRANSACTION [ READ WRITE | READ ONLY ];

**SQL Select**

The most commonly used SQL command is **SELECT statement**. It is used to query the database and retrieve selected data that follow the conditions we want.

Let's see the syntax of select statement.

1.    **SELECT** expressions

2.    **FROM** tables

3.    **WHERE** conditions;

Here expression is the column that we want to retrieve.

Tables indicate the tables, we want to retrieve records from.

Optional clauses in SELECT statement

There are some optional clauses in SELECT statement:

**[WHERE Clause]** : It specifies which rows to retrieve.

**[GROUP BY Clause]** : Groups rows that share a property so that the aggregate function can be applied to each group.

**[HAVING Clause]** : It selects among the groups defined by the GROUP BY clause.

**[ORDER BY Clause]** : It specifies an order in which to return the rows.

For example, let a database table: student\_details;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **First\_name** | **Last\_name** | **Age** | **Subject** | **Hobby** |
| 1 | Amar | Sharma | 20 | Maths | Cricket |
| 2 | Akbar | Khan | 22 | Biology | Football |
| 3 | Anthony | Milton | 25 | Commerce | Gambling |

From the above example, select the first name of all the students. To do so, query should be like this:

1.    **SELECT** first\_name **FROM** student\_details;

Note: the SQL commands are not case sensitive. We can also write the above SELECT statement as:

1.    **select** first\_name **from** student\_details;

Now, you will get following data:

|  |
| --- |
| Amar |
| Akbar |
| Anthony |

We can also retrieve data from more than one column. For example, to select first name and last name of all the students, you need to write

1.    **SELECT** first\_name, last\_name **FROM** student\_details;

Now, you will get following data:

|  |  |
| --- | --- |
| Amar | Sharma |
| Akbar | Khan |
| Anthony | Milton |

We can also use clauses like WHERE, GROUP BY, HAVING, ORDER BY with SELECT statement.

Here a point is notable that only SELECT and FROM statements are necessary in SQL SELECT statements. Other clauses like WHERE, GROUP BY, ORDER BY, HAVING may be optional.

SQL SELECT DISTINCT

The **SQL DISTINCT command** is used with SELECT key word to retrieve only distinct or unique data.

In a table, there may be a chance to exist a duplicate value and sometimes we want to retrieve only unique values. In such scenarios, SQL SELECT DISTINCT statement is used.

**Note: SQL SELECT UNIQUE and SQL SELECT DISTINCT statements are same.**

Let's see the syntax of select distinct statement.

1.    **SELECT** **DISTINCT** column\_name ,column\_name

2.    **FROM**  table\_name;

Let's try to understand it by the table given below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Student\_Name** | **Gender** | **Mobile\_Number** | **HOME\_TOWN** |
| Rahul Ojha | Male | 7503896532 | Lucknow |
| Disha Rai | Female | 9270568893 | Varanasi |
| Sonoo Jaiswal | Male | 9990449935 | Lucknow |

Here is a table of students from where we want to retrieve distinct information For example: distinct home-town.

1.    **SELECT** **DISTINCT** home\_town

2.    **FROM** students

Now, it will return two rows.

|  |
| --- |
| **HOME\_TOWN** |
| Lucknow |
| Varanasi |

SQL SELECT TOP

The SQL SELECT TOP Statement is used to select top data from a table. The top clause specifies that how many rows are returned.

Let's see an example. If a table has a large number of data, select top statement determines that how many rows will be retrieved from the given table.

There is an example of employee table:

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_ID** | **NAME** | **SIR\_NAME** | **USER\_NAME** |
| 1 | RAHUL | OJHA | ra@jha |
| 2 | ANU | SHARMA | anusha1 |
| 3 | RAVI | SINGHAL | Ravin |

Let's see the syntax for the select top statement.

1.    **SELECT** COUNT (expression)

Let's see the example of sql select top statement.

1.    **SELECT** **TOP** 2 \* **FROM** employee

It will return the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_ID** | **NAME** | **SIR\_NAME** | **USER\_NAME** |
| 1 | RAHUL | OJHA | ra@jha |
| 2 | ANU | SHARMA | anusha1 |

SQL SELECT AS

**SQL AS** is used to assign temporarily a new name to a table column.

It makes easy presentation of query results and allows the developer to label results more accurately without permanently renaming table columns.

Let's see the example of select as:

1.    **SELECT** day\_of\_order **AS** "Date"

2.    Customer **As** "Client",

3.    Product,

4.    Quantity,

5.    **FROM** orders;

Let us take a table named orders, it contains:

|  |  |  |  |
| --- | --- | --- | --- |
| **Day\_of\_order** | **Customer** | **Product** | **Quantity** |
| 11-09-2001 | Osama bin qasim | Airplane | 2 |
| 13-12-2001 | Mukhtar mahmood | RDX | 20 |
| 26-12-2004 | Balaswamy | Water cannon | 35 |

After applying this SQL AS example syntax

1.    **SELECT** day\_of\_order **AS** "Date"

2.    Customer **As** "Client",

3.    Product,

4.    Quantity,

5.    **FROM** orders;

Result will be shown as this table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Client** | **Product** | **Quantity** |
| 11-09-2001 | Osama bin qasim | airplane | 2 |
| 13-12-2001 | Mukhtar mahmood | RDX | 20 |
| 26-12-2004 | Balaswamy | Water cannon | 35 |

Note: SQL AS is same as SQL ALIAS

SQL SELECT IN

SQL IN is an operator used in a SQL query to help reduce the need to use multiple SQL "OR" conditions.

It is used in SELECT, INSERT, UPDATE or DELETE statement.

Advantage of SQL SELECT IN

It minimizes the use of SQL OR operator.

Let's see the syntax for SQL IN:

1.    Expression IN (value 1, value 2 ... value n);

Take an example with character values.

1.    **SELECT** \*

2.    **FROM** students

3.    **WHERE** students\_name IN ( Amit , Raghav, Rajeev)

Let’s take another example with numeric values.

1.    **SELECT** \*

2.    **FROM** marks

3.    **WHERE** roll\_no IN (001, 023, 024);

SQL SELECT from Multiple Tables

This statement is used to retrieve fields from multiple tables. To do so, we need to use join query to get data from multiple tables.

Let's see the example for the select from multiple tables:

1.    **SELECT** orders.order\_id, suppliers.**name**

2.    **FROM** suppliers

3.    **INNER** JOIN orders

4.    **ON** suppliers.supplier\_id = orders.supplier\_id

5.    **ORDER** **BY** order\_id;

Let us take three tables, two tables of customers named customer1 and customer2 and the third table is product table.

**Customer1 table**

|  |  |
| --- | --- |
| **Cus\_id** | **Name1** |
| 1 | Jack |
| 2 | Jill |

**Customer2 table**

|  |  |
| --- | --- |
| **Cus\_id** | **Name2** |
| 1 | Sandy |
| 2 | Venus |

**Product table**

|  |  |  |
| --- | --- | --- |
| **P\_id** | **Cus\_id** | **P\_name** |
| 1 | 1 | Laptop |
| 2 | 2 | Phone |
| 3 | P1 | Pen |
| 4 | P2 | Notebook |

**Example syntax to select from multiple tables:**

1.    **SELECT** p. p\_id, p.cus\_id, p.p\_name, c1.name1, c2.name2

2.    **FROM** product **AS** p

3.    LEFT JOIN customer1 **AS** c1

4.    **ON** p.cus\_id=c1.cus\_id

5.    LEFT JOIN customer2 **AS** c2

6.    **ON** p.cus\_id = c2.cus\_id

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_id** | **Cus\_id** | **P\_name** | **P\_name** | **P\_name** |
| 1 | 1 | Laptop | Jack | NULL |
| 2 | 2 | Phone | Jill | NULL |
| 3 | P1 | Pen | NULL | Sandy |
| 4 | P2 | Notebook | NULL | Venus |

SQL SELECT INTO Statement

The SELECT INTO statement selects data from one table and inserts it into a new table.

SQL SELECT INTO Syntax

We can copy all columns into the new table:

SELECT \*  
INTO *newtable* [IN *externaldb*]  
FROM *table1;*

Or we can copy only the columns we want into the new table:

SELECT *column\_name(s)*  
INTO *newtable* [IN *externaldb*]  
FROM *table1;*

The new table will be created with the column-names and types as defined in the SELECT statement. You can apply new names using the AS clause.

SQL SELECT INTO Examples

Create a backup copy of Customers:

SELECT \*  
INTO CustomersBackup2013  
FROM Customers;

Use the IN clause to copy the table into another database:

SELECT \*  
INTO CustomersBackup2013 IN 'Backup.mdb'  
FROM Customers;

Copy only a few columns into the new table:

SELECT CustomerName, ContactName  
INTO CustomersBackup2013  
FROM Customers;

Copy only the German customers into the new table:

SELECT \*  
INTO CustomersBackup2013  
FROM Customers  
WHERE Country='Germany';

Copy data from more than one table into the new table:

SELECT Customers.CustomerName, Orders.OrderID  
INTO CustomersOrderBackup2013  
FROM Customers  
LEFT JOIN Orders  
ON Customers.CustomerID=Orders.CustomerID;

SQL INSERT INTO SELECT Statement

The INSERT INTO SELECT statement selects data from one table and inserts it into an existing table. Any existing rows in the target table are unaffected.

SQL INSERT INTO SELECT Syntax

We can copy all columns from one table to another, existing table:

INSERT INTO *table2*  
SELECT \* FROM *table1;*

Or we can copy only the columns we want to into another, existing table:

INSERT INTO *table2*  
*(column\_name(s))*  
SELECT *column\_name(s)*  
FROM *table1;*

SQL INSERT INTO SELECT Examples

Copy only a few columns from "Suppliers" into "Customers":

Example

INSERT INTO Customers (CustomerName, Country)  
SELECT SupplierName, Country FROM Suppliers;

Copy only the German suppliers into "Customers":

Example

INSERT INTO Customers (CustomerName, Country)  
SELECT SupplierName, Country FROM Suppliers  
WHERE Country='Germany';

**SQL Clauses**

SQL WHERE

A **WHERE clause** in SQL is a data manipulation language statement.

WHERE clauses are not mandatory clauses of SQL DML statements. But it can be used to limit the number of rows affected by a SQL DML statement or returned by a query.

Actually, it filters the records. It returns only those queries which fulfill the specific conditions.

WHERE clause is used in SELECT, UPDATE, DELETE statement etc.

Let's see the syntax for sql where:

1.    **SELECT** column1, **column** 2, ... **column** n

2.    **FROM**    table\_name

3.    **WHERE** [conditions]

WHERE clause uses some conditional selection

|  |  |
| --- | --- |
| = | Equal |
| > | greater than |
| < | less than |
| >= | greater than or equal |
| <= | less than or equal |
| < > | not equal to |

SQL AND

The SQL AND condition is used in SQL query to create two or more conditions to be met.

It is used in SQL SELECT, INSERT, UPDATE and DELETE statements.

Let's see the syntax for SQL AND:

1.    **SELECT** columns

2.    **FROM** tables

3.    **WHERE** condition 1

4.    AND condition 2;

The SQL AND condition requires that both conditions should be met.

The SQL AND condition also can be used to join multiple tables in a SQL statement.

SQL "AND" example with "INSERT" statement

This is how an SQL "AND" condition can be used in the SQL INSERT statement.

For example:

1.    **INSERT** **INTO** suppliers

2.    (supplier\_id, supplier\_name)

3.    **SELECT** account\_no, **name**

4.    **FROM** customers

5.    **WHERE** customer\_name ='IBM'

6.    AND employees =1000;

SQL "AND" example with "UPDATE" statement

This is how the "AND" condition can be used in the SQL UPDATE statement.

For example:

1.    **UPDATE** suppliers

2.    **SET** supplier\_name = 'HP'

3.    **WHERE** supplier\_name = 'IBM'

4.    AND offices = 8;

SQL "AND" example with "DELETE" statement

This is how an SQL "AND" condition can be used in the SQL DELETE statement.

For example:

1.    **DELETE** **FROM** suppliers

2.    **WHERE** supplier\_name = 'IBM'

3.    AND product = 'PC computers';

SQL OR

The **SQL OR condition** is used in a SQL query to create a SQL statement where records are returned when any one of the condition met. It can be used in a SELECT statement, INSERT statement, UPDATE statement or DELETE statement.

Let's see the syntax for OR condition:

1.    **SELECT** columns

2.    **FROM** tables

3.    **WHERE** condition 1

4.    OR condition 2;

SQL "OR" example with SQL SELECT

1.    **SELECT** \*

2.    **FROM** suppliers

3.    **WHERE** city = 'New York'

4.    OR available\_products >= 250;

SQL "OR" example with SQL INSERT

You can see in below example that how an SQL "OR" condition is used with SQL insert statement.

For example:

1.    **INSERT** **INTO** suppliers(supplier\_id, supplier\_name)

2.    **SELECT** account\_no, **name**

3.    **FROM** customers

4.    **WHERE** city = 'New Delhi'

5.    OR city = 'Ghaziabad';

SQL "OR" example with SQL UPDATE

For example:

1.    **UPDATE** suppliers

2.    **SET** supplier\_name = 'HP'

3.    **WHERE** supplier\_name = 'IBM'

4.    OR available\_product >36;

SQL "OR" example with SQL DELETE

For example:

1.    **DELETE** **FROM** suppliers

2.    **WHERE** supplier\_name = 'IBM'

3.    OR employee <=100;

SQL BETWEEN

The BETWEEN operator selects values within a range. The values can be numbers, text, or dates.

You can use **BETWEEN** clause to replace a combination of "greater than equal AND less than equal" conditions.

SQL BETWEEN Syntax

SELECT *column\_name(s)*  
FROM *table\_name*  
WHERE *column\_name*BETWEEN *value1* AND *value2;*

To understand **BETWEEN**clause, consider an **employee\_tbl** table, which is having following records:

mysql> SELECT \* FROM employee\_tbl;

+------+------+------------+--------------------+

| id   | name | work\_date  | daily\_typing\_pages |

+------+------+------------+--------------------+

|    1 | John | 2007-01-24 |                250 |

|    2 | Ram  | 2007-05-27 |                220 |

|    3 | Jack | 2007-05-06 |                170 |

|    3 | Jack | 2007-04-06 |                100 |

|    4 | Jill | 2007-04-06 |                220 |

|    5 | Zara | 2007-06-06 |                300 |

|    5 | Zara | 2007-02-06 |                350 |

+------+------+------------+--------------------+

7 rows in set (0.00 sec)

Now, suppose based on the above table you want to fetch records with conditions daily\_typing\_pages more than 170 and equal and less than 300 and equal. This can be done using **>=** and **<=** conditions as follows:

mysql>SELECT \* FROM employee\_tbl

    ->WHERE daily\_typing\_pages >= 170 AND

    ->daily\_typing\_pages <= 300;

+------+------+------------+--------------------+

| id   | name | work\_date  | daily\_typing\_pages |

+------+------+------------+--------------------+

|    1 | John | 2007-01-24 |                250 |

|    2 | Ram  | 2007-05-27 |                220 |

|    3 | Jack | 2007-05-06 |                170 |

|    4 | Jill | 2007-04-06 |                220 |

|    5 | Zara | 2007-06-06 |                300 |

+------+------+------------+--------------------+

5 rows in set (0.03 sec)

Same can be achieved using **BETWEEN** clause as follows:

mysql> SELECT \* FROM employee\_tbl

    -> WHERE daily\_typing\_pages BETWEEN 170 AND 300;

+------+------+------------+--------------------+

| id   | name | work\_date  | daily\_typing\_pages |

+------+------+------------+--------------------+

|    1 | John | 2007-01-24 |                250 |

|    2 | Ram  | 2007-05-27 |                220 |

|    3 | Jack | 2007-05-06 |                170 |

|    4 | Jill | 2007-04-06 |                220 |

|    5 | Zara | 2007-06-06 |                300 |

+------+------+------------+--------------------+

NOT BETWEEN Operator Example

To display the products outside the range of the previous example, use NOT BETWEEN:

Example

SELECT \* FROM Products  
WHERE Price NOT BETWEEN 10 AND 20;

BETWEEN Operator with IN Example

The following SQL statement selects all products with a price BETWEEN 10 and 20, but products with a CategoryID of 1,2, or 3 should not be displayed:

Example

SELECT \* FROM Products  
WHERE (Price BETWEEN 10 AND 20)  
AND NOT CategoryID IN (1,2,3);

NOT BETWEEN Operator with Text Value Example

The following SQL statement selects all products with a ProductName beginning with any of the letter NOT BETWEEN 'C' and 'M':

Example

SELECT \* FROM Products  
WHERE ProductName NOT BETWEEN 'C' AND 'M';

BETWEEN Operator with Date Value Example

The following SQL statement selects all orders with an OrderDate BETWEEN '04-July-1996' and '09-July-1996':

Example

SELECT \* FROM Orders  
WHERE OrderDate BETWEEN #07/04/1996# AND #07/09/1996#;

SQL - LIKE Clause

The SQL **LIKE** clause is used to compare a value to similar values using wildcard operators. There are two wildcards used in conjunction with the LIKE operator:

        The percent sign (%)

        The underscore (\_)

The percent sign represents zero, one, or multiple characters. The underscore represents a single number or character. The symbols can be used in combinations.

Syntax:

The basic syntax of % and \_ is as follows:

SELECT FROM table\_name

WHERE column LIKE 'XXXX%'

or

SELECT FROM table\_name

WHERE column LIKE '%XXXX%'

or

SELECT FROM table\_name

WHERE column LIKE 'XXXX\_'

or

SELECT FROM table\_name

WHERE column LIKE '\_XXXX'

or

SELECT FROM table\_name

WHERE column LIKE '\_XXXX\_'

You can combine N number of conditions using AND or OR operators. Here, XXXX could be any numeric or string value.

Example:

Here are number of examples showing WHERE part having different LIKE clause with '%' and '\_' operators:

|  |  |
| --- | --- |
| **Statement** | **Description** |
| WHERE SALARY LIKE '200%' | Finds any values that start with 200 |
| WHERE SALARY LIKE '%200%' | Finds any values that have 200 in any position |
| WHERE SALARY LIKE '\_00%' | Finds any values that have 00 in the second and third positions |
| WHERE SALARY LIKE '2\_%\_%' | Finds any values that start with 2 and are at least 3 characters in length |
| WHERE SALARY LIKE '%2' | Finds any values that end with 2 |
| WHERE SALARY LIKE '\_2%3' | Finds any values that have a 2 in the second position and end with a 3 |
| WHERE SALARY LIKE '2\_\_\_3' | Finds any values in a five-digit number that start with 2 and end with 3 |

Let us take a real example, consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| ID | NAME     | AGE | ADDRESS   | SALARY   |

+----+----------+-----+-----------+----------+

|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |

|  2 | Khilan   |  25 | Delhi     |  1500.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

|  4 | Chaitali |  25 | Mumbai    |  6500.00 |

|  5 | Hardik   |  27 | Bhopal    |  8500.00 |

|  6 | Komal    |  22 | MP        |  4500.00 |

|  7 | Muffy    |  24 | Indore    | 10000.00 |

+----+----------+-----+-----------+----------+

Following is an example, which would display all the records from CUSTOMERS table where SALARY starts with 200:

SQL> SELECT \* FROM CUSTOMERS

WHERE SALARY LIKE '200%';

This would produce the following result:

+----+----------+-----+-----------+----------+

| ID | NAME     | AGE | ADDRESS   | SALARY   |

+----+----------+-----+-----------+----------+

|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

**SQL Wildcards**

The SQL wildcards can be used to search data within a table.

SQL wildcards are used with SQL LIKE operator.

The boolean NOT operator in the select statement can be used as wildcard NOT LIKE operator.

In SQL, the wildcards are :

|  |  |
| --- | --- |
| **Wildcards** | **Description** |
| % | The percent sign character (%) represent a sequence of 0 (zero) or more characters. |
| Underscore ( \_ ) | The underscore character ( \_ ) represents a single character. |
| [charlist] | It represents any single character within a charlist |
| [^charlist] or [!charlist] | It represents any single character other than the charlist |

SQL - TOP, LIMIT or ROWNUM Clause

The SQL **TOP** clause is used to fetch a TOP N number or X percent records from a table.

**Note:** All the databases do not support TOP clause. For example MySQL supports **LIMIT** clause to fetch limited number of records and Oracle uses **ROWNUM** to fetch limited number of records.

Syntax:

The basic syntax of TOP clause with SELECT statement would be as follows:

SELECT TOP number|percent column\_name(s)

FROM table\_name

WHERE [condition]

Example:

Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| ID | NAME     | AGE | ADDRESS   | SALARY   |

+----+----------+-----+-----------+----------+

|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |

|  2 | Khilan   |  25 | Delhi     |  1500.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

|  4 | Chaitali |  25 | Mumbai    |  6500.00 |

|  5 | Hardik   |  27 | Bhopal    |  8500.00 |

|  6 | Komal    |  22 | MP        |  4500.00 |

|  7 | Muffy    |  24 | Indore    | 10000.00 |

+----+----------+-----+-----------+----------+

Following is an example on SQL server, which would fetch top 3 records from CUSTOMERS table:

SQL> SELECT TOP 3 \* FROM CUSTOMERS;

This would produce the following result:

+----+---------+-----+-----------+---------+

| ID | NAME    | AGE | ADDRESS   | SALARY  |

+----+---------+-----+-----------+---------+

|  1 | Ramesh  |  32 | Ahmedabad | 2000.00 |

|  2 | Khilan  |  25 | Delhi     | 1500.00 |

|  3 | kaushik |  23 | Kota      | 2000.00 |

+----+---------+-----+-----------+---------+

If you are using MySQL server, then here is an equivalent example:

SQL> SELECT \* FROM CUSTOMERS

LIMIT 3;

This would produce the following result:

+----+---------+-----+-----------+---------+

| ID | NAME    | AGE | ADDRESS   | SALARY  |

+----+---------+-----+-----------+---------+

|  1 | Ramesh  |  32 | Ahmedabad | 2000.00 |

|  2 | Khilan  |  25 | Delhi     | 1500.00 |

|  3 | kaushik |  23 | Kota      | 2000.00 |

+----+---------+-----+-----------+---------+

If you are using Oracle server, then here is an equivalent example:

SQL> SELECT \* FROM CUSTOMERS

WHERE ROWNUM <= 3;

This would produce the following result:

+----+---------+-----+-----------+---------+

| ID | NAME    | AGE | ADDRESS   | SALARY  |

+----+---------+-----+-----------+---------+

|  1 | Ramesh  |  32 | Ahmedabad | 2000.00 |

|  2 | Khilan  |  25 | Delhi     | 1500.00 |

|  3 | kaushik |  23 | Kota      | 2000.00 |

+----+---------+-----+-----------+---------+

SQL - Having

The HAVING clause enables you to specify conditions that filter which group results appear in the final results.

The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause.

Syntax:

The following is the position of the HAVING clause in a query:

SELECT

FROM

WHERE

GROUP BY

HAVING

ORDER BY

The HAVING clause must follow the GROUP BY clause in a query and must also precede the ORDER BY clause if used. The following is the syntax of the SELECT statement, including the HAVING clause:

SELECT column1, column2

FROM table1, table2

WHERE [ conditions ]

GROUP BY column1, column2

HAVING [ conditions ]

ORDER BY column1, column2

Example:

Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| ID | NAME     | AGE | ADDRESS   | SALARY   |

+----+----------+-----+-----------+----------+

|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |

|  2 | Khilan   |  25 | Delhi     |  1500.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

|  4 | Chaitali |  25 | Mumbai    |  6500.00 |

|  5 | Hardik   |  27 | Bhopal    |  8500.00 |

|  6 | Komal    |  22 | MP        |  4500.00 |

|  7 | Muffy    |  24 | Indore    | 10000.00 |

+----+----------+-----+-----------+----------+

Following is the example, which would display record for which similar age count would be more than or equal to 2:

SQL > SELECT ID, NAME, AGE, ADDRESS, SALARY

FROM CUSTOMERS

GROUP BY age

HAVING COUNT(age) >= 2;

This would produce the following result:

+----+--------+-----+---------+---------+

| ID | NAME   | AGE | ADDRESS | SALARY  |

+----+--------+-----+---------+---------+

|  2 | Khilan |  25 | Delhi   | 1500.00 |

+----+--------+-----+---------+---------+

**SQL Aggregate functions**

There are various aggregate functions that can be used with SELECT statement. A list of commonly used aggregate functions are given below.

count() Function

The MySQL count() function is used to return the count of an expression. It is used when you need to count some records of your table.

**Syntax:**

**SELECT** COUNT (aggregate\_expression)

**FROM** table\_name

[**WHERE** conditions];

Parameter explanation

**aggregate\_expression:** It specifies the column or expression whose NON-NULL values will be counted.

**table\_name:** It specifies the tables, from where you want to retrieve records. There must be at least one table listed in the FROM clause.

**WHERE conditions:** It is optional. It specifies the conditions that must be fulfilled for the records to be selected.

Example:

**SELECT** COUNT(officer\_name)

**FROM** officers;

sum() function

The MySQL sum() function is used to return the total summed value of an expression.

**Syntax:**

**SELECT** SUM(aggregate\_expression)

**FROM** tables

[**WHERE** conditions];

**Example:**

**SELECT** SUM (working\_hours) **AS** "Total working hours"

**FROM** employees

**WHERE** working\_hours > 5;

avg() function

The MySQL avg() function is used to return the average value of an expression.

**Syntax:**

**SELECT** AVG(aggregate\_expression)

**FROM** tables

[**WHERE** conditions];

Example:

**SELECT** AVG(working\_hours) **AS** "Avg working hours"

**FROM** employees

**WHERE** working\_hours > 5;

min() function

The MySQL min() function is used to return the minimum value from the table.

**Syntax:**

**SELECT** **MIN** (aggregate\_expression)

**FROM** tables

[**WHERE** conditions];

Example:

**SELECT** **MIN** (working\_hours) **AS** "Minimum working hours"

**FROM** employees;

max() function

The MySQL max() function is used to return the maximum value of an expression. It is used when you need to get the maximum value from your table.

**Syntax:**

**SELECT** **MAX**(aggregate\_expression)

**FROM** tables

[**WHERE** conditions];

Example:

**SELECT** **MAX** (working\_hours) **AS** "Maximum working hours"

**FROM** employees;

first function

The MySQL first function is used to return the first value of the selected column. Here, we use limit clause to select first record or more.

**Syntax:**

**SELECT** column\_name

**FROM** table\_name

LIMIT 1;

Example:

**SELECT** officer\_name

**FROM** officers

LIMIT 1;

**To SELECT FIRST two records**

**SELECT** officer\_name

**FROM** officers

LIMIT 2;

last function

MySQL last function is used to return the last value of the selected column.

**Syntax:**

**SELECT** column\_name

**FROM** table\_name

**ORDER** **BY** column\_name **DESC**

LIMIT 1;

Example:

**SELECT** officer\_name

**FROM** officers

**ORDER** **BY** officer\_id **DESC**

LIMIT 1;

**SQL Sub queries and SQL Injection**

SQL - Sub Queries

A Subquery or Inner query or 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 cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY 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   |  35 | Ahmedabad |  2000.00 |

|  2 | Khilan   |  25 | Delhi     |  1500.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

|  4 | Chaitali |  25 | Mumbai    |  6500.00 |

|  5 | Hardik   |  27 | Bhopal    |  8500.00 |

|  6 | Komal    |  22 | MP        |  4500.00 |

|  7 | Muffy    |  24 | Indore    | 10000.00 |

+----+----------+-----+-----------+----------+

Now, let us check following subquery with SELECT statement:

SQL> SELECT \*

     FROM CUSTOMERS

     WHERE ID IN (SELECT ID

                  FROM CUSTOMERS

                  WHERE SALARY > 4500) ;

This would produce the following result:

+----+----------+-----+---------+----------+

| ID | NAME     | AGE | ADDRESS | SALARY   |

+----+----------+-----+---------+----------+

|  4 | Chaitali |  25 | Mumbai  |  6500.00 |

|  5 | Hardik   |  27 | Bhopal  |  8500.00 |

|  7 | Muffy    |  24 | Indore  | 10000.00 |

+----+----------+-----+---------+----------+

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 complete CUSTOMERS table into CUSTOMERS\_BKP, following is the 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.

Following example updates SALARY by 0.25 times in 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   |  35 | Ahmedabad |   125.00 |

|  2 | Khilan   |  25 | Delhi     |  1500.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

|  4 | Chaitali |  25 | Mumbai    |  6500.00 |

|  5 | Hardik   |  27 | Bhopal    |  2125.00 |

|  6 | Komal    |  22 | MP        |  4500.00 |

|  7 | Muffy    |  24 | Indore    | 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 CUSTOMERS\_BKP table available which is backup of CUSTOMERS table.

Following example deletes records from 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 CUSTOMERS table would have the following records:

+----+----------+-----+---------+----------+

| ID | NAME     | AGE | ADDRESS | SALARY   |

+----+----------+-----+---------+----------+

|  2 | Khilan   |  25 | Delhi   |  1500.00 |

|  3 | kaushik  |  23 | Kota    |  2000.00 |

|  4 | Chaitali |  25 | Mumbai  |  6500.00 |

|  6 | Komal    |  22 | MP      |  4500.00 |

|  7 | Muffy    |  24 | Indore  | 10000.00 |

+----+----------+-----+---------+----------+

SQL Injection

SQL injection is a technique where malicious users can inject SQL commands into an SQL statement, via web page input.

Injected SQL commands can alter SQL statement and compromise the security of a web application.

SQL Injection Based on 1=1 is Always True

Look at the example above, one more time.

Let's say that the original purpose of the code was to create an SQL statement to select a user with a given user id.

If there is nothing to prevent a user from entering "wrong" input, the user can enter some "smart" input like this:

UserId:   


Server Result

SELECT \* FROM Users WHERE UserId = 105 or 1=1;

The SQL above is valid. It will return all rows from the table Users, since **WHERE 1=1** is always true.

Does the example above seem dangerous? What if the Users table contains names and passwords?

The SQL statement above is much the same as this:

SELECT UserId, Name, Password FROM Users WHERE UserId = 105 or 1=1;

A smart hacker might get access to all the user names and passwords in a database by simply inserting 105 or 1=1 into the input box.

SQL Injection Based on ""="" is Always True

Here is a common construction, used to verify user login to a web site:

User Name:  


Password:  


Server Code

uName = getRequestString("UserName");  
uPass = getRequestString("UserPass");  
  
sql = 'SELECT \* FROM Users WHERE Name ="' + uName + '" AND Pass ="' + uPass + '"'

Result

SELECT \* FROM Users WHERE Name ="John Doe" AND Pass ="myPass"

A smart hacker might get access to user names and passwords in a database by simply inserting " or ""=" into the user name or password text box:

User Name:  


Password:  


The code at the server will create a valid SQL statement like this:

Result

SELECT \* FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""

The result SQL is valid. It will return all rows from the table Users, since **WHERE ""=""** is always true.

SQL Injection Based on Batched SQL Statements

Most databases support batched SQL statement, separated by semicolon.

Example

SELECT \* FROM Users; DROP TABLE Suppliers

The SQL above will return all rows in the Users table, and then delete the table called Suppliers.

If we had the following server code:

Server Code

txtUserId = getRequestString("UserId");  
txtSQL = "SELECT \* FROM Users WHERE UserId = " + txtUserId;

And the following input:

User id:  


The code at the server would create a valid SQL statement like this:

Result

SELECT \* FROM Users WHERE UserId = 105; DROP TABLE Suppliers

SQL> DELETE FROM CUSTOMERS

     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   |

+----+----------+-----+---------+----------+

|  2 | Khilan   |  25 | Delhi   |  1500.00 |

|  3 | kaushik  |  23 | Kota    |  2000.00 |

|  4 | Chaitali |  25 | Mumbai  |  6500.00 |

|  6 | Komal    |  22 | MP      |  4500.00 |

|  7 | Muffy    |  24 | Indore  | 10000.00 |

### 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**;

#### 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*

#### 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.

#### 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 |

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

  CREATE VIEW MarksView AS

  SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS

  FROM StudentDetails, StudentMarks

  WHERE StudentDetails.NAME = StudentMarks.NAME;

To display data of View MarksView:

SELECT \* FROM MarksView;

Output:

**DELETING VIEWS**

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

**Syntax**:

DROP VIEW view\_name;

**view\_name**: Name of the View which we want to delete.

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

DROP VIEW MarksView;

**UPDATING VIEWS**

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

1.    The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.

2.    The SELECT statement should not have the DISTINCT keyword.

3.    The View should have all NOT NULL values.

4.    The view should not be created using nested queries or complex queries.

5.    The view should be created from a single table. If the view is created using multiple tables then we will not be allowed to update the view.

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

  CREATE OR REPLACE VIEW view\_name AS

  SELECT column1,coulmn2,..

  FROM table\_name

  WHERE condition;

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

CREATE OR REPLACE VIEW MarksView AS

SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS, StudentMarks.AGE

FROM StudentDetails, StudentMarks

WHERE StudentDetails.NAME = StudentMarks.NAME;

If we fetch all the data from MarksView now as:

SELECT \* FROM MarksView;

Output:

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

  INSERT view\_name(column1, column2 , column3,..)

  VALUES(value1, value2, value3..);



  **view\_name**: Name of the View

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

INSERT INTO DetailsView(NAME, ADDRESS)

VALUES("Suresh","Gurgaon");

If we fetch all the data from DetailsView now as,

SELECT \* FROM DetailsView;

Output:

  **Deleting a row from a View**:  
Deleting rows from a view is also as simple as deleting rows from a table. We can use the DELETE statement of SQL to delete rows from a view. Also deleting a row from a view first delete the row from the actual table and the change is then reflected in the view.**Syntax**:

  DELETE FROM view\_name

  WHERE condition;



  **view\_name**:Name of view from where we want to delete rows

  **condition**: Condition to select rows

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

DELETE FROM DetailsView

WHERE NAME="Suresh";

If we fetch all the data from DetailsView now as,

SELECT \* FROM DetailsView;

Output:

**WITH CHECK OPTION**

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

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

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

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

CREATE VIEW SampleView AS

SELECT S\_ID, NAME

FROM  StudentDetails

WHERE NAME IS NOT NULL

WITH CHECK OPTION;

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

INSERT INTO SampleView(S\_ID)

VALUES(6);

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

# Materialized Views in Oracle

A materialized view, or snapshot as they were previously known, is a table segment whose contents are periodically refreshed based on a query, either against a local or remote table. Using materialized views against remote tables is the simplest way to achieve replication of data between sites.

## Basic Syntax

The full syntax description for the [CREATE MATERIALIZED VIEW](http://docs.oracle.com/cd/E11882_01/server.112/e41084/statements_6002.htm) command is available in the documentation. Here we will only concern ourselves with the basics.

-- Normal

CREATE MATERIALIZED VIEW view-name

BUILD [IMMEDIATE | DEFERRED]

REFRESH [FAST | COMPLETE | FORCE ]

ON [COMMIT | DEMAND ]

[[ENABLE | DISABLE] QUERY REWRITE]

AS

SELECT ...;

The BUILD clause options are shown below.

* IMMEDIATE : The materialized view is populated immediately.
* DEFERRED : The materialized view is populated on the first requested refresh.

The following refresh types are available.

* FAST : A fast refresh is attempted. If materialized view logs are not present against the source tables in advance, the creation fails.
* COMPLETE : The table segment supporting the materialized view is truncated and repopulated completely using the associated query.
* FORCE : A fast refresh is attempted. If one is not possible a complete refresh is performed.

A refresh can be triggered in one of two ways.

* ON COMMIT : The refresh is triggered by a committed data change in one of the dependent tables.
* ON DEMAND : The refresh is initiated by a manual request or a scheduled task.

The QUERY REWRITE clause tells the optimizer if the materialized view should be consider for query rewrite operations. An example of the query rewrite functionality is shown below.

The ON PREBUILT TABLE clause tells the database to use an existing table segment, which must have the same name as the materialized view and support the same column structure as the query.

Create Materialized View

Connect to the materialized view owner and create the database link and the materialized view itself.

CONNECT scott/tiger@db2

CREATE DATABASE LINK DB1.WORLD CONNECT TO scott IDENTIFIED BY tiger USING 'DB1.WORLD';

CREATE MATERIALIZED VIEW emp\_mv

BUILD IMMEDIATE

REFRESH FORCE

ON DEMAND

AS

SELECT \* FROM emp@db1.world;

Refresh Materialized Views

If a materialized view is configured to refresh on commit, you should never need to manually refresh it, unless a rebuild is necessary. Remember, refreshing on commit is a very intensive operation for volatile base tables. It makes sense to use fast refreshes where possible.

For on demand refreshes, you can choose to manually refresh the materialized view or refresh it as part of a refresh group.

## Difference between View vs Materialized View in database

Based upon on our understanding of View and Materialized View, Let’s see, some short difference between them :

1) First difference between View and materialized view is that, In Views query **result is not stored in the disk** or database but Materialized view allow to store query result in disk or table.

2) Another difference between View vs materialized view is that, when we create view using any table,  rowid of view is same as original table but in case of Materialized view rowid is different.

3) One more *difference between View and materialized view in database* is that, In case of View we always get latest data but in case of Materialized view we need to refresh the view for getting latest data.

4) Performance of View is less than Materialized view.

5) This is continuation of first difference between View and Materialized View, In case of view its only the logical view of table no separate copy of table but in case of Materialized view we get physically separate copy of table

6) Last difference between View vs Materialized View is that, In case of Materialized view we need extra trigger or some automatic method so that we can keep MV refreshed, this is not required for views in database.

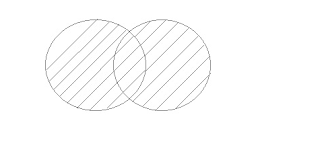
**SQL Set Operations and AUTO-INCREMENT**

**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.

**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 datatype must be same in both the tables.

[](https://4.bp.blogspot.com/-81oN4z4AUbw/WMoH9d6y86I/AAAAAAAACzQ/YzB1mR3c1HcQZUEm4uwVyMp41JXhMAtBQCLcB/s1600/Untitled.png)

**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**

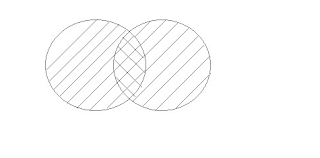
select \* from second

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.

[](https://1.bp.blogspot.com/-E6Bw5h5ifm8/WMoI4jYETUI/AAAAAAAACzY/xRiLelZTh48fBw8CrzFQ5Fgy_j1NC7l0ACLcB/s1600/Untitled.png)

**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**

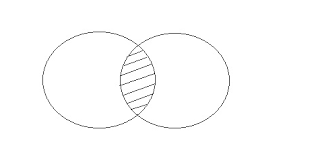
select \* from second

The result table will look like,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | Abhi |
| 2 | Adam |
| 2 | Adam |
| 3 | Chester |

**Intersect**

Intersect operation is used to combine two SELECT statements, but it only retuns the records which are common from both SELECT statements. In case of **Intersect** the number of columns and datatype must be same. MySQL does not support INTERSECT operator.

[](https://3.bp.blogspot.com/-8D2-dzP0hyk/WMoJl6hfKkI/AAAAAAAACzg/s8D6j2LPchchg83Tp0mp1LcVz4CRQnHwACLcB/s1600/Untitled.png)

**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**

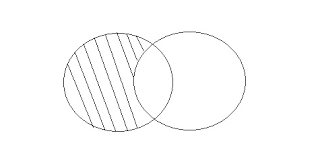
select \* from second

The result table will look like

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 2 | Adam |

**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.

[](https://4.bp.blogspot.com/-ISa7CqaS2XE/WMoJ70FaYhI/AAAAAAAACzk/yIEIQ4cLnZ80N0tb2pRppGeKZr5cnoyNQCLcB/s1600/Untitled.png)

**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 |

In short,

UNION - the values of the first query are returned with the values of the second query eliminating duplicates.  
MINUS - the values of the first query are returned with duplicates values of the second query removed from the first query.  
UNION ALL - the values of both queries are returned including all duplicates  
INTERSECT - only the duplicate values are returned from both queries.

SQL AUTO INCREMENT

Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table.

Often this is the primary key field that we would like to be created automatically every time a new record is inserted.

Syntax for MySQL

The following SQL statement defines the "ID" column to be an auto-increment primary key field in the "Persons" table:

CREATE TABLE Persons (  
    ID int NOT NULL AUTO\_INCREMENT,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    PRIMARY KEY (ID)  
);

MySQL uses the AUTO\_INCREMENT keyword to perform an auto-increment feature.

By default, the starting value for AUTO\_INCREMENT is 1, and it will increment by 1 for each new record.

To let the AUTO\_INCREMENT sequence start with another value, use the following SQL statement:

ALTER TABLE Persons AUTO\_INCREMENT=100;

To insert a new record into the "Persons" table, we will NOT have to specify a value for the "ID" column (a unique value will be added automatically):

INSERT INTO Persons (FirstName,LastName)  
VALUES ('Lars','Monsen');

The SQL statement above would insert a new record into the "Persons" table. The "ID" column would be assigned a unique value. The "FirstName" column would be set to "Lars" and the "LastName" column would be set to "Monsen".

**SQL JOIN**

SQL JOIN

As the name shows, JOIN means *to combine something*. In case of SQL, JOIN means **"to combine two or more tables"**.  
The SQL JOIN clause takes records from two or more tables in a database and combines it together.  
**ANSI standard SQL** defines five types of JOIN :

1. inner join,
2. left outer join,
3. right outer join,
4. full outer join, and
5. cross join.

In the process of joining, rows of both tables are combined in a single table.

Why SQL JOIN is used?

If you want to access more than one table through a select statement.  
If you want to combine two or more table then SQL JOIN statement is used .it combines rows of that tables in one table and one can retrieve the information by a SELECT statement.  
The joining of two or more tables is based on common field between them.  
SQL INNER JOIN also known as simple join is the most common type of join.

How to use SQL join or SQL Inner Join?

Let an example to deploy SQL JOIN process:  
1.Staff table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Staff\_NAME** | **Staff\_AGE** | **STAFF\_ADDRESS** | **Monthley\_Package** |
| 1 | ARYAN | 22 | MUMBAI | 18000 |
| 2 | SUSHIL | 32 | DELHI | 20000 |
| 3 | MONTY | 25 | MOHALI | 22000 |
| 4 | AMIT | 20 | ALLAHABAD | 12000 |

2.Payment table

|  |  |  |  |
| --- | --- | --- | --- |
| **Payment\_ID** | **DATE** | **Staff\_ID** | **AMOUNT** |
| 101 | 30/12/2009 | 1 | 3000.00 |
| 102 | 22/02/2010 | 3 | 2500.00 |
| 103 | 23/02/2010 | 4 | 3500.00 |

So if you follow this JOIN statement to join these two tables ?

1.   **SELECT** Staff\_ID, Staff\_NAME, Staff\_AGE, AMOUNT

2.      **FROM** STAFF s, PAYMENT p

3.      **WHERE** s.ID =p.STAFF\_ID;

This will produce the result like this:

|  |  |  |  |
| --- | --- | --- | --- |
| **STAFF\_ID** | **NAME** | **Staff\_AGE** | **AMOUNT** |
| 3 | MONTY | 25 | 2500 |
| 1 | ARYAN | 22 | 3000 |
| 4 | AMIT | 25 | 3500 |
|  |  |  |  |

SQL LEFT JOIN

The SQL left join returns all the values from the left table and it also includes matching values from right table, if there are no matching join value it returns NULL.

**BASIC SYNTAX FOR LEFT JOIN:**

1.    **SELECT** table1.column1, table2.column2....

2.    **FROM** table1

3.    LEFTJOIN table2

4.    **ON** table1.column\_field = table2.column\_field;

let us take two tables in this example to elaborate all the things:

**CUSTOMER TABLE:**

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **SALARY** |
| 1 | ARYAN | 51 | 56000 |
| 2 | AROHI | 21 | 25000 |
| 3 | VINEET | 24 | 31000 |
| 4 | AJEET | 23 | 32000 |
| 5 | RAVI | 23 | 42000 |

This is second table

**ORDER TABLE:**

|  |  |  |  |
| --- | --- | --- | --- |
| **O\_ID** | **DATE** | **CUSTOMER\_ID** | **AMOUNT** |
| 001 | 20-01-2012 | 2 | 3000 |
| 002 | 12-02-2012 | 2 | 2000 |
| 003 | 22-03-2012 | 3 | 4000 |
| 004 | 11-04-2012 | 4 | 5000 |

**join these two tables with LEFT JOIN:**

1.    SQL **SELECT** ID, **NAME**, AMOUNT,**DATE**

2.    **FROM** CUSTOMER

3.    LEFT JOIN **ORDER**

4.    **ON** CUSTOMER.ID = **ORDER**.CUSTOMER\_ID;

This will produce the following result:

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **AMOUNT** | **DATE** |
| 1 | ARYAN | NULL | NULL |
| 2 | AROHI | 3000 | 20-01-2012 |
| 2 | AROHI | 2000 | 12-02-2012 |
| 3 | VINEET | 4000 | 22-03-2012 |
| 4 | AJEET | 5000 | 11-04-2012 |
| 5 | RAVI | NULL | NULL |

SQL RIGHT JOIN

The SQL right join returns all the values from the rows of right table. It also includes the matched values from left table but if there is no matching in both tables, it returns NULL.

Basic syntax for right join:

1.    **SELECT** table1.column1, table2.column2.....

2.    **FROM** table1

3.    RIGHT JOIN table2

4.    **ON** table1.column\_field = table2.column\_field;

let us take an example with 2 tables table1 is CUSTOMERS table and table2 is ORDERS table.

**CUSTOMER TABLE:**

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **SALARY** |
| 1 | ARYAN | 51 | 56000 |
| 2 | AROHI | 21 | 25000 |
| 3 | VINEET | 24 | 31000 |
| 4 | AJEET | 23 | 32000 |
| 5 | RAVI | 23 | 42000 |

and this is the second table:

**ORDER TABLE:**

|  |  |  |  |
| --- | --- | --- | --- |
| **DATE** | **O\_ID** | **CUSTOMER\_ID** | **AMOUNT** |
| 20-01-2012 | 001 | 2 | 3000 |
| 12-02-2012 | 002 | 2 | 2000 |
| 22-03-2012 | 003 | 3 | 4000 |
| 11-04-2012 | 004 | 4 | 5000 |

Here we will join these two tables with SQL RIGHT JOIN:

1.    SQL> **SELECT** ID,**NAME**,AMOUNT,**DATE**

2.    **FROM** CUSTOMER

3.    RIGHT JOIN **ORDER**

4.    **ON** CUSTOMER.ID = **ORDER**.CUSTOMER\_ID;

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **AMOUNT** | **DATE** |
| 2 | AROHI | 3000 | 20-01-2012 |
| 2 | AROHI | 2000 | 12-02-2012 |
| 3 | VINEET | 4000 | 22-03-2012 |
| 4 | AJEET | 5000 | 11-04-2012 |

SQL FULL JOIN

The SQL full join is the result of combination of both left and right outer join and the join tables have all the records from both tables. It puts NULL on the place of matches not found.

SQL full outer join and SQL join are same. generally it is known as SQL FULL JOIN.

SQL full outer join:

What is SQL full outer join?

SQL full outer join is used to combine the result of both left and right outer join and returns all rows (don?t care its matched or unmatched) from the both participating tables.

**Syntax for full outer join:**

1.    **SELECT** \*

2.    **FROM** table1

3.    **FULL** OUTER JOIN table2

4.    **ON** table1.column\_name = table2.column\_name;

**Note:**here table1 and table2 are the name of the tables participating in joining and column\_name is the column of the participating tables.

Let us take two tables to demonstrate full outer join:

**table\_A**

|  |  |
| --- | --- |
| **A** | **M** |
| 1 | m |
| 2 | n |
| 4 | o |

**table\_B**

|  |  |
| --- | --- |
| **A** | **N** |
| 2 | p |
| 3 | q |
| 5 | r |

**Resulting table**

|  |  |  |  |
| --- | --- | --- | --- |
| **A** | **M** | **A** | **N** |
| 2 | n | 2 | p |
| 1 | m | - | - |
| 4 | o | - | - |
| - | - | 3 | q |
| - | - | 5 | r |

Because this is a full outer join so all rows (both matching and non-matching) from both tables are included in the output. Here only one row of output displays values in all columns because there is only one match between table\_A and table\_B.

### SQL DML(Insert,Update,Delete)

# SQL INSERT STATEMENT

SQL INSERT statement is a SQL query. It is used to insert a single or a multiple records in a table.

There are two ways to insert data in a table:

1. By SQL insert into statement
   1. By specifying column names
   2. Without specifying column names
2. By SQL insert into select statement

## 1) Inserting data directly into a table

You can insert a row in the table by using SQL INSERT INTO command. But there are 2 ways to do this.

You can specify or ignore the column names while using INSERT INTO statement.

To insert partial column values, you must have to specify the column names. But if you want to insert all the column values, you can specify or ignore the column names.

If you specify the column names, syntax of the insert into statement will be as follows:

1.    **INSERT** **INTO** TABLE\_NAME

2.    [(col1, col2, col3,.... col N)]

3.    **VALUES** (value1, value2, value 3, .... Value N);

Here col1, col2, col3, .... colN are the columns of the table in which you want to insert data.

##### Note: At the time of inserting a row into table, if you add values for all columns then there is no need to specify the column name in SQL INSERT query. Moreover, you must be sure that you are entering the values in the same order as the columns exist.

But, If you ignore the column names, syntax of the insert into statement will be as follows:

1.    **INSERT** **INTO** TABLE\_NAME

2.    **VALUES** (value1, value2, value 3, .... Value N);

## 2) Inserting data through SELECT Statement

**SQL INSERT INTO SELECT Syntax**

1.    **INSERT** **INTO** table\_name

2.    [(column1, column2, .... **column**)]

3.    **SELECT** column1, column2, .... **Column** N

4.    **FROM** table\_name [**WHERE** condition];

**Note:** when you add a new row, you should make sure that data type of the value and the column should be matched.

If any integrity constraints are defined for the table, you must follow them.

# SQL INSERT INTO VALUE

There are two ways to insert values in a table.

**In the first method there is no need to specify the column name where the data will be inserted, you need only their values.**

1.    **INSERT** **INTO** table\_name

2.    **VALUES** (value1, value2, value3....);

**The second method specifies both the column name and values which you want to insert.**

1.    **INSERT** **INTO** table\_name (column1, column2, column3....)

2.    **VALUES** (value1, value2, value3.....);

# SQL UPDATE

The SQL commands (*UPDATE* and *DELETE*) are used to modify the data that is already in the database. The SQL DELETE command uses a WHERE clause.

**SQL UPDATE** statement is used to change the data of the records held by tables. Which rows is to be update, it is decided by a condition. To specify condition, we use WHERE clause.

The UPDATE statement can be written in following form:

1.    **UPDATE** table\_name **SET** [column\_name1= value1,... column\_nameN = value N] [**WHERE** condition]

Let's see the Syntax:

1.    **UPDATE** table\_name

2.    **SET** column\_name = expression

3.    **WHERE** conditions

Let's take an example: here we are going to update an entry in the source table.

SQL statement:

1.    **UPDATE** students

2.    **SET** User\_Name = 'beinghuman'

3.    **WHERE** Student\_Id = '3'

**Source Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Student\_Id** | **FirstName** | **LastName** | **User\_Name** |
| 1 | Ada | Sharma | sharmili |
| 2 | Rahul | Maurya | sofamous |
| 3 | James | Walker | jonny |

See the result after updating value:

|  |  |  |  |
| --- | --- | --- | --- |
| **Student\_Id** | **FirstName** | **LastName** | **User\_Name** |
| 1 | Ada | Sharma | sharmili |
| 2 | Rahul | Maurya | sofamous |
| 3 | James | Walker | **beinghuman** |

## 

## Updating Multiple Fields:

If you are going to update multiple fields, you should separate each field assignment with a comma.

SQL UPDATE statement for multiple fields:

1.    **UPDATE** students

2.    **SET** User\_Name = 'beserious', First\_Name = 'Johnny'

3.    **WHERE** Student\_Id = '3'

Result of the table is given below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Student\_Id** | **FirstName** | **LastName** | **User\_Name** |
| 1 | Ada | Sharma | sharmili |
| 2 | Rahul | Maurya | Sofamous |
| 3 | **Johnny** | Walker | **beserious** |

MYSQL SYNTAX FOR UPDATING TABLE:

1.    **UPDATE** table\_name

2.    **SET** field1 = new-value1, field2 = new-value2,

3.    [**WHERE** CLAUSE]

**SQL UPDATE SELECT:**

**SQL UPDATE WITH SELECT QUERY:**

We can use SELECT statement to update records through UPDATE statement.

**SYNTAX:**

1.    **UPDATE** tableDestination

2.    **SET** tableDestination.col = value

3.    **WHERE** EXISTS (

4.    **SELECT** col2.value

5.    **FROM**  tblSource

6.    **WHERE** tblSource.join\_col = tblDestination. Join\_col

7.    AND  tblSource.**Constraint** = value)

You can also try this one -

1.    **UPDATE**

2.    **Table**

3.    **SET**

4.    **Table**.column1 = other table.**column** 1,

5.    **Table**.column2 = other table.**column** 2

6.    **FROM**

7.    **Table**

8.    **INNER** JOIN

9.    Other\_table

10. **ON**

11. **Table**.id = other\_table.id

**My SQL SYNTAX:**

If you want to UPDATE with SELECT in My SQL, you can use this syntax:

Let's take an example having two tables. Here,

First table contains -

Cat\_id, cat\_name,

And the second table contains -

Rel\_cat\_id, rel\_cat\_name

**SQL UPDATE COLUMN:**

We can update a single or multiple columns in SQL with SQL UPDATE query.

**SQL UPDATE EXAMPLE WITH UPDATING SINGLE COLUMN:**

1.    **UPDATE** students

2.    **SET** student\_id = 001

3.    **WHERE** student\_name = 'AJEET';

This SQL UPDATE example would update the student\_id to '001' in the student table where student\_name is 'AJEET'.

**SQL UPDATE EXAMPLE WITH UPDATING MULTIPLE COLUMNS:**

To update more than one column with a single update statement:

1.    **UPDATE** students

2.    **SET** student\_name = 'AJEET',

3.    Religion = 'HINDU'

4.    **WHERE** student\_name = 'RAJU';

This SQL UPDATE statement will change the student name to 'AJEET' and religion to 'HINDU' where the student name is 'RAJU'.

# SQL UPDATE with JOIN

**SQL UPDATE JOIN** means we will update one table using another table and join condition.

### How to use multiple tables in SQL UPDATE statement with JOIN

Let's take two tables, table 1 and table 2

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Col 1** | **Col 2** | **Col 3** |
| 1 | 1 | 11 | First |
| 2 | 11 | 12 | Second |
| 3 | 21 | **13** | **Third** |
| 4 | 31 | **14** | **Fourth** |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Col 1** | **Col 2** | **Col 3** |
| 1 | 1 | 21 | Two-One |
| 2 | 11 | 22 | Two-Two |
| 3 | 21 | **23** | **Two-Three** |
| 4 | 31 | **24** | **Two-Four** |

Our requirement is that we have table 2 which has two rows where Col 1 is 21 and 31. We want to update the value from table 2 to table 1 for the rows where Col 1 is 21 and 31.

We want to also update the values of Col 2 and Col 3 only.

The most easiest and common way is to use join clause in the update statement and use multiple tables in the update statement.

1.    **UPDATE** **table** 1

2.    **SET** Col 2 = t2.Col2,

3.    Col 3 = t2.Col3

4.    **FROM** table1 t1

5.    **INNER** JOIN **table** 2 t2 **ON** t1.Col1 = t2.col1

6.    **WHERE** t1.Col1 IN (21,31)

Check the content of the table

SELECT FROM table 1

SELECT FROM table 2

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Col 1** | **Col 2** | **Col 3** |
| 1 | 1 | 11 | First |
| 2 | 11 | 12 | Second |
| 3 | 21 | **23** | **Two-Three** |
| 4 | 31 | **24** | **Two-Four** |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Col 1** | **Col 2** | **Col 3** |
| 1 | 1 | 21 | First |
| 2 | 11 | 22 | Second |
| 3 | 21 | **23** | **Two-Three** |
| 4 | 31 | **24** | **Two-Four** |

Here we can see that using join clause in update statement. We have merged two tables by the use of join clause.

# SQL UPDATE DATE

How to update a date and time field in SQL?  
If you want to update a date & time field in SQL, you should use the following query.  
let's see the syntax of sql update date.

1.   **UPDATE** **table**

2.   **SET** Column\_Name = 'YYYY-MM-DD HH:MM:SS'

3.   **WHERE** Id = value

Let us check this by an example:  
Firstly we take a table in which we want to update date and time fields.  
If you want to change the first row which id is 1 then you should write the following syntax:

1.   **UPDATE** **table**

2.   **SET** EndDate = '2014-03-16 00:00:00.000'

3.   **WHERE** Id = 1

<Note: you should always remember that SQL must attach default 00:00:00.000 automatically.

This query will change the date and time field of the first row in that above assumed table.

# SQL DELETE

The **SQL DELETE statement** is used to delete rows from a table. Generally DELETE statement removes one or more records from a table.

#### SQL DELETE Syntax

Let's see the Syntax for the SQL DELETE statement:

1.    **DELETE** **FROM** table\_name [**WHERE** condition];

Here table\_name is the table which has to be deleted. The *WHERE clause* in SQL DELETE statement is optional here.

# SQL DELETE TABLE

The DELETE statement is used to delete rows from a table. If you want to remove a specific row from a table you should use WHERE condition.

1.    **DELETE** **FROM** table\_name [**WHERE** condition];

But if you do not specify the WHERE condition it will remove all the rows from the table.

1.    **DELETE** **FROM** table\_name;

# SQL DELETE ROW

Let us take an example of student.

**Original table:**

|  |  |  |
| --- | --- | --- |
| **ID** | **STUDENT \_NAME** | **ADDRESS** |
| 001 | AJEET MAURYA | GHAZIABAD |
| 002 | RAJA KHAN | LUCKNOW |
| 003 | RAVI MALIK | DELHI |

If you want to delete a student with id 003 from the student\_name table, then the SQL DELETE query should be like this:

1.    **DELETE** **FROM** student\_name

2.    **WHERE** id = 003;

Resulting table after SQL DELETE query:

|  |  |  |
| --- | --- | --- |
| **ID** | **STUDENT\_NAME** | **ADDRESS** |
| 001 | AJEET MAURYA | GHAZIABAD |
| 002 | RAJA KHAN | LUCKNOW |

# SQL DELETE ALL ROWS

The statement SQL DELETE ALL ROWS is used to delete all rows from the table. If you want to delete all the rows from student table the query would be like,

1.    **DELETE** **FROM** STUDENT\_NAME;

# SQL DELETE DUPLICATE ROWS

If you have got a situation that you have multiple duplicate records in a table, so at the time of fetching records from the table you should be more careful. You make sure that you are fetching unique records instead of fetching duplicate records.

To overcome with this problem we use DISTINCT keyword.

It is used along with SELECT statement to eliminate all duplicate records and fetching only unique records.

**SYNTAX:**

The basic syntax to eliminate duplicate records from a table is:

1.    **SELECT** **DISTINCT** column1, column2,....columnN

2.    **FROM** **table** \_name

3.    **WHERE** [conditions]

**SQL ORDER BY & GROUP BY Clause**

SQL ORDER BY Clause

The SQL ORDER BY clause is used for sorting data in ascending and descending order based on one or more columns.

Some databases sort query results in ascending order by default.

SQL ORDER BY syntax:

1.    **SELECT** expressions

2.    **FROM** tables

3.    **WHERE** conditions

4.    **ORDER** **BY** expression [**ASC** | **DESC**];

Let us take a CUSTOMERS table having the following records

|  |
| --- |
|  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Himani gupta | 21 | Modinagar | 22000 |
| 2 | Shiva tiwari | 22 | Bhopal | 21000 |
| 3 | Ajeet bhargav | 45 | Meerut | 65000 |
| 4 | Ritesh yadav | 36 | Azamgarh | 26000 |
| 5 | Balwant singh | 45 | Varanasi | 36000 |
| 6 | Mahesh Sharma | 26 | Mathura | 22000 |

This is an example that would sort the result in ascending order by NAME and SALARY.

1.    **SELECT** \* **FROM** CUSTOMERS

2.    **ORDER** **BY** **NAME**, SALARY;

This would produce the following result.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 3 | Ajeet bhargav | 45 | Meerut | 65000 |
| 5 | Balwant singh | 45 | Varanasi | 36000 |
| 1 | Himani gupta | 21 | Modinagar | 22000 |
| 6 | Mahesh Sharma | 26 | Mathura | 22000 |
| 4 | Ritesh yadav | 36 | Azamgarh | 26000 |
| 2 | Shiva tiwari | 22 | Bhopal | 21000 |

This is an example to sort the result in descending order by NAME.

1.    **SELECT** \* **FROM** CUSTOMERS

2.    **ORDER** **BY** **NAME** **DESC**;

This would produce the following result.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 2 | Shiva tiwari | 22 | Bhopal | 21000 |
| 4 | Ritesh yadav | 36 | Azamgarh | 26000 |
| 6 | Mahesh Sharma | 26 | Mathura | 22000 |
| 1 | Himani gupta | 21 | Modinagar | 22000 |
| 5 | Balwant singh | 45 | Varanasi | 36000 |
| 3 | Ajeet bhargav | 45 | Meerut | 65000 |

SQL ORDER BY CLAUSE WITH ASCENDING ORDER

This statement is used to sort data in ascending order. If you miss the ASC attribute, SQL ORDER BY query takes ascending order by default.

Let us take a CUSTOMERS table having the following records:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Himani gupta | 21 | Modinagar | 22000 |
| 2 | Shiva tiwari | 22 | Bhopal | 21000 |
| 3 | Ajeet bhargav | 45 | Meerut | 65000 |
| 4 | Ritesh yadav | 36 | Azamgarh | 26000 |
| 5 | Balwant singh | 45 | Varanasi | 36000 |
| 6 | Mahesh Sharma | 26 | Mathura | 22000 |

This is an example to sort the result in ascending order by NAME and SALARY.

1.    **SELECT** \* **FROM** CUSTOMERS

2.    **ORDER** **BY** **NAME**, SALARY;

This would produce the following result.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 3 | Ajeet bhargav | 45 | Meerut | 65000 |
| 5 | Balwant singh | 45 | Varanasi | 36000 |
| 1 | Himani gupta | 21 | Modinagar | 22000 |
| 6 | Mahesh Sharma | 26 | Mathura | 22000 |
| 4 | Ritesh yadav | 36 | Azamgarh | 26000 |
| 2 | Shiva tiwari | 22 | Bhopal | 21000 |

SQL ORDER BY CLAUSE WITH DESCENDING ORDER:

This statement is used to sort data in descending order. You should use the DESC attribute in your ORDER BY clause as follows.

1.    **SELECT** supplier\_city

2.    **FROM** suppliers

3.    **WHERE** supplier\_name = 'IBM'

4.    **ORDER** **BY** supplier\_city **DESC**;

Let's see an example of an employee table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Himani gupta | 21 | Modinagar | 22000 |
| 2 | Shiva tiwari | 22 | Bhopal | 21000 |
| 3 | Ajeet bhargav | 45 | Meerut | 65000 |
| 4 | Ritesh yadav | 36 | Azamgarh | 26000 |
| 5 | Balwant singh | 45 | Varanasi | 36000 |
| 6 | Mahesh Sharma | 26 | Mathura | 22000 |

This is an example to sort the result in descending order by NAME.

1.    **SELECT** \* **FROM** CUSTOMERS

2.    **ORDER** **BY** **NAME** **DESC**;

This would produce the following result.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 2 | Shiva tiwari | 22 | Bhopal | 21000 |
| 4 | Ritesh yadav | 36 | Azamgarh | 26000 |
| 6 | Mahesh sharma | 26 | Mathura | 22000 |
| 1 | Himani gupta | 21 | Modinagar | 22000 |
| 5 | Balwant singh | 45 | Varanasi | 36000 |
| 3 | Ajeet bhargav | 45 | Meerut | 65000 |

SQL SORTING ON MULTIPLE COLUMNS

Let's take an example of customer table which has many columns, the following SQL statement selects all customers from the table named "customer", stored by the "country" and "Customer-Name" columns:

1.    **SELECT** \* **FROM** customers

2.    **ORDER** **BY** country, Customer-**Name**;

SQL - Group By

The SQL **GROUP BY**clause is used in collaboration with the SELECT statement to arrange identical data into groups.

The GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

Syntax:

The basic syntax of GROUP BY clause is given below. The GROUP BY clause must follow the conditions in the WHERE clause and must precede the ORDER BY clause if one is used.

SELECT column1, column2

FROM table\_name

WHERE [ conditions ]

GROUP BY column1, column2

ORDER BY column1, column2

Example:

Consider the CUSTOMERS table is having the following records:

+----+----------+-----+-----------+----------+

| ID | NAME     | AGE | ADDRESS   | SALARY   |

+----+----------+-----+-----------+----------+

|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |

|  2 | Khilan   |  25 | Delhi     |  1500.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

|  4 | Chaitali |  25 | Mumbai    |  6500.00 |

|  5 | Hardik   |  27 | Bhopal    |  8500.00 |

|  6 | Komal    |  22 | MP        |  4500.00 |

|  7 | Muffy    |  24 | Indore    | 10000.00 |

+----+----------+-----+-----------+----------+

If you want to know the total amount of salary on each customer, then GROUP BY query would be as follows:

SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS

     GROUP BY NAME;

This would produce the following result:

+----------+-------------+

| NAME     | SUM(SALARY) |

+----------+-------------+

| Chaitali |     6500.00 |

| Hardik   |     8500.00 |

| kaushik  |     2000.00 |

| Khilan   |     1500.00 |

| Komal    |     4500.00 |

| Muffy    |    10000.00 |

| Ramesh   |     2000.00 |

+----------+-------------+

Now, let us have following table where CUSTOMERS table has the following records with duplicate names:

+----+----------+-----+-----------+----------+

| ID | NAME     | AGE | ADDRESS   | SALARY   |

+----+----------+-----+-----------+----------+

|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |

|  2 | Ramesh   |  25 | Delhi     |  1500.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

|  4 | kaushik  |  25 | Mumbai    |  6500.00 |

|  5 | Hardik   |  27 | Bhopal    |  8500.00 |

|  6 | Komal    |  22 | MP        |  4500.00 |

|  7 | Muffy    |  24 | Indore    | 10000.00 |

+----+----------+-----+-----------+----------+

Now again, if you want to know the total amount of salary on each customer, then GROUP BY query would be as follows:

SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS

     GROUP BY NAME;

This would produce the following result:

+---------+-------------+

| NAME    | SUM(SALARY) |

+---------+-------------+

| Hardik  |     8500.00 |

| kaushik |     8500.00 |

| Komal   |     4500.00 |

| Muffy   |    10000.00 |

| Ramesh  |     3500.00 |

+---------+-------------+

### SQL String & Date Functions

# SQL - String Functions

SQL string functions are used primarily for string manipulation. The following table details the important string functions:

 ASCII(str)

Returns the numeric value of the leftmost character of the string str. Returns 0 if str is the empty string. Returns NULL if str is NULL. ASCII() works for characters with numeric values from 0 to 255.

>SELECT ASCII('4');

52

>SELECT ASCII('2');

50

BIN(N)

Returns a string representation of the binary value of N, where N is a longlong (BIGINT) number. This is equivalent to CONV(N,10,2). Returns NULL if N is NULL.

>SELECT BIN(13);

1101

>SELECT BIN(12);

1100

BIT\_LENGTH(str)

Returns the length of the string str in bits.

>SELECT BIT\_LENGTH('text');

32

>SELECT BIT\_LENGTH('texte');

40

>SELECT BIT\_LENGTH('texter');

48

CHAR(N,... [USING charset\_name])

CHAR() interprets each argument N as an integer and returns a string consisting of the characters given by the code values of those integers. NULL values are skipped.

SELECT CHAR(77,121,83,81,'76');

MySQL

>SELECT CHAR(65,97,83,82,'86');

AaSRV

  (A to Z) -> (65 to 90) respectively       (a to z)-> (97 to 122) respectively

CHAR\_LENGTH(str)

Returns the length of the string str measured in characters. A multi-byte character counts as a single character. This means that for a string containing five two-byte characters, LENGTH() returns 10, whereas CHAR\_LENGTH() returns 5.

>SELECT CHAR\_LENGTH("text");

4

>SELECT CHAR\_LENGTH("texte");

5

CHARACTER\_LENGTH(str)

CHARACTER\_LENGTH() is a synonym for CHAR\_LENGTH().

CONCAT(str1,str2,...)

Returns the string that results from concatenating the arguments. May have one or more arguments. If all arguments are non-binary strings, the result is a non-binary string. If the arguments include any binary strings, the result is a binary string.

>SELECT CONCAT('My', 'S', 'QL');

MySQL

CONCAT\_WS(separator,str1,str2,...)

CONCAT\_WS() stands for Concatenate With Separator and is a special form of CONCAT().

>SELECT CONCAT\_WS(',','First name','Last Name' );

  First name,Last name

CONV(N,from\_base,to\_base)

Converts numbers between different number bases. Returns a string representation of the number N, converted from base from\_base to to\_base. Returns NULL if any argument is NULL.

>SELECT CONV('a',16,2);

1010

>SELECT CONV(15,10,2);

1111

>SELECT CONV(11,10,2);

1011

ELT(N,str1,str2,str3,...)

Returns str1 if N = 1, str2 if N = 2, and so on. Returns NULL if N is less than 1 or greater than the number of arguments. ELT() is the complement of FIELD().

>SELECT ELT(1, 'ej', 'Heja', 'hej', 'foo');

ej

>SELECT ELT(3, 'ej', 'Heja', 'hej', 'foo');

hej

>SELECT ELT(5, 'ej', 'Heja', 'hej', 'foo');

NULL

EXPORT\_SET(bits,on,off[,separator[,number\_of\_bits]])

Returns a string such that for every bit set in the value bits, you get an on string and for every bit not set in the value, you get an off string. Bits in bits are examined from right to left (from low-order to high-order bits). Strings are added to the result from left to right, separated by the separator string (the default being the comma character .,.).

>SELECT EXPORT\_SET(5,'Y','N',',',4);

Y,N,Y,N

FIELD(str,str1,str2,str3,...)

Returns the index (position starting with 1) of str in the str1, str2, str3, ... list. Returns 0 if str is not found.

>SELECT FIELD('ej', 'Hej', 'ej', 'ejh', 'hej', 'foo')

2

>SELECT FIELD('ej', 'Hej', 'egj', 'ejh', 'ej', 'foo')

4

FIND\_IN\_SET(str,strlist)

Returns a value in the range of 1 to N if the string str is in the string list strlist consisting of N substrings.

>SELECT FIND\_IN\_SET('b','a,c,b,d');

3

>SELECT FIND\_IN\_SET('c','c,a,b,d');

1

FORMAT(X,D)

Formats the number X to a format like '#,###,###.##', rounded to D decimal places, and returns the result as a string. If D is 0, the result has no decimal point or fractional part.

>SELECT FORMAT(12332.123456, 4);

12,332.1235

>SELECT FORMAT(12332.123456, 3);

12,332.123

>SELECT FORMAT(12332.123456, 0);

12,332

HEX(N\_or\_S)

If N\_or\_S is a number, returns a string representation of the hexadecimal value of N, where N is a longlong (BIGINT) number. This is equivalent to CONV(N,10,16).

If N\_or\_S is a string, returns a hexadecimal string representation of N\_or\_S where each character in N\_or\_S is converted to two hexadecimal digits.

>SELECT HEX(255);

FF

>SELECT 0x646566;

def

>SELECT 646566;

646566

INSERT(str,pos,len,newstr)

Returns the string str, with the substring beginning at position pos and len characters long replaced by the string newstr. Returns the original string if pos is not within the length of the string. Replaces the rest of the string from position pos if len is not within the length of the rest of the string. Returns NULL if any argument is NULL.

>SELECT INSERT('Quadratic', 3, 4, 'What');

QuWhattic

>SELECT INSERT('Quadratic', 2, 1, 'What');

Qwhatadratic

INSTR(str,substr)

Returns the position of the first occurrence of substring substr in string str. This is the same as the two-argument form of LOCATE(), except that the order of the arguments is reversed.

>SELECT INSTR('foobarbar', 'bar');

4

>SELECT INSTR('foobarbar', 'bar');

3

LCASE(str)

LCASE() is a synonym for LOWER().

LEFT(str,len)

Returns the leftmost len characters from the string str, or NULL if any argument is NULL.

>SELECT LEFT('foobarbar', 6);

foobar

>SELECT LEFT('foobarbar', 1);

f

LENGTH(str)

Returns the length of the string str, measured in bytes

>SELECT LENGTH('text');

4

>SELECT LENGTH('textert');

7

LOCATE(substr,str), LOCATE(substr,str,pos)

The first syntax returns the position of the first occurrence of substring substr in string str. The second syntax returns the position of the first occurrence of substring substr in string str, starting at position pos. Returns 0 if substr is not in str.

>SELECT LOCATE('bar', 'fbarbar');

2

>SELECT LOCATE('bar', 'foobarbar');

4

LOWER(str)

Returns the string str with all characters changed to lowercase according to the current character set mapping.

>SELECT LOWER('QUADRATICALLY');

quadratically

LPAD(str,len,padstr)

Returns the string str, left-padded with the string padstr to a length of len characters. If str is longer than len, the return value is shortened to len characters.

>SELECT LPAD('hiartff',4,'??');

hiar

>SELECT LPAD('hia',4,'??');

?hia

>SELECT LPAD('hi',4,'??');

??hi

LTRIM(str)

Returns the string str with leading space characters removed.

>SELECT LTRIM('   barbar');

barbar

MAKE\_SET(bits,str1,str2,...)

Returns a set value (a string containing substrings separated by .,. characters) consisting of the strings that have the corresponding bit in bits set. str1 corresponds to bit 0, str2 to bit 1, and so on. NULL values in str1, str2, ... are not appended to the result.

>SELECT MAKE\_SET(1,'a','b','c');

a

>SELECT MAKE\_SET(b,'a','b','c');

B

>SELECT MAKE\_SET(3,'a','b','c');

a,b

>SELECT MAKE\_SET(7,'a','b','c');

a,b,c

MID(str,pos,len)

MID(str,pos,len) is a synonym for SUBSTRING(str,pos,len).

OCT(N)

Returns a string representation of the octal value of N, where N is a longlong (BIGINT) number. This is equivalent to CONV(N,10,8). Returns NULL if N is NULL.

>SELECT OCT(12);

14

>SELECT OCT(26);

32

OCTET\_LENGTH(str)

OCTET\_LENGTH() is a synonym for LENGTH().

ORD(str)

If the leftmost character is not a multi-byte character, ORD() returns the same value as the ASCII() function.

>SELECT ORD('4');

52

>SELECT ORD('2’);

50

POSITION(substr IN str)

POSITION(substr IN str) is a synonym for LOCATE(substr,str).

QUOTE(str)

Quotes a string to produce a result that can be used as a properly escaped data value in an SQL statement. The string is returned enclosed by single quotes and with each instance of single quote ('), backslash ('\'), ASCII NUL, and Control-Z preceded by a backslash. If the argument is NULL, the return value is the word 'NULL' without enclosing single quotes.

>SELECT QUOTE('Don\\\'t!');

‘Don\\\’t!’

expr REGEXP pattern

This function performs a pattern match of expr against pattern. Returns 1 if expr matches pat; otherwise it returns 0. If either expr or pat is NULL, the result is NULL. REGEXP is not case sensitive, except when used with binary strings.

>SELECT 'ABCDEF' REGEXP 'A%C%%';

0

>SELECT 'ABCDE' REGEXP '.\*';

1

>SELECT 'new\*\n\*line' REGEXP 'new\\\*.\\\*line';

1

REPEAT(str,count)

Returns a string consisting of the string str repeated count times. If count is less than 1, returns an empty string. Returns NULL if str or count are NULL.

>SELECT REPEAT('SQL', 3);

SQLSQLSQL

REPLACE(str,from\_str,to\_str)

Returns the string str with all occurrences of the string from\_str replaced by the string to\_str. REPLACE() performs a case-sensitive match when searching for from\_str.

>SELECT REPLACE('www.mysql.com', 'm', 'Ww');

www.Wwysql.coWw

REVERSE(str)

Returns the string str with the order of the characters reversed.

>SELECT REVERSE('abcd');

dcba

RIGHT(str,len)

Returns the rightmost len characters from the string str, or NULL if any argument is NULL.

>SELECT RIGHT('foobarbar', 6);

barbar

RPAD(str,len,padstr)

Returns the string str, right-padded with the string padstr to a length of len characters. If str is longer than len, the return value is shortened to len characters.

>SELECT RPAD('hi',5,'?');

hi???

>SELECT RPAD('hiaa',3,'?');

hia

RTRIM(str)

Returns the string str with trailing space characters removed.

>SELECT RTRIM('barbar   ');

barbar

SOUNDEX(str)

Returns a soundex string from str. A standard soundex string is four characters long, but the SOUNDEX() function returns an arbitrarily long string. You can use SUBSTRING() on the result to get a standard soundex string.

>SELECT SOUNDEX('Hello');

H400

>SELECT SOUNDEX('friend');

F653

expr1 SOUNDS LIKE expr2

This is the same as SOUNDEX(expr1) = SOUNDEX(expr2).

SPACE(N)

Returns a string consisting of N space characters.

>SELECT SPACE(2);

‘     ‘

STRCMP(str1, str2)

Compares two strings and returns 0 if both strings are equal, it returns -1 if the first argument is smaller than the second according to the current sort order otherwise it returns 1.

>SELECT STRCMP('MOHD', 'MNHD');

1

>SELECT STRCMP('MOHD', 'MOHD');

0

>SELECT STRCMP('AMOHD', 'MOHD');

-1

>SELECT STRCMP('MOHD', 'AMOHD');

1

SUBSTRING(str,pos)

>SELECT SUBSTRING('Quadratically',5);

radically

SUBSTRING(str FROM pos)

>SELECT SUBSTRING('foobarbar' FROM 2);

obarbar

SUBSTRING(str,pos,len)

>SELECT SUBSTRING('Quadratically',3,4);

adra

SUBSTRING\_INDEX(str,delim,count)

Returns the substring from string str before count occurrences of the delimiter delim. If count is positive, everything to the left of the final delimiter (counting from the left) is returned. If count is negative, everything to the right of the final delimiter (counting from the right) is returned. SUBSTRING\_INDEX() performs a case-sensitive match when searching for delim.

>SELECT SUBSTRING\_INDEX('www.mysql.com', '.', 2);

[www.mysql](http://www.mysql/)

TRIM([{BOTH | LEADING | TRAILING} [remstr] FROM] str) SELECT TRIM(TRAILING 'xyz' FROM 'barxxyz');

Returns the string str with all remstr prefixes or suffixes removed. If none of the specifiers BOTH, LEADING, or TRAILING is given, BOTH is assumed. remstr is optional and, if not specified, spaces are removed.

>SELECT TRIM('  bar   ');

bar

>SELECT TRIM(LEADING 'x' FROM 'xxxbarxxx');

barxxx

>SELECT TRIM(BOTH 'x' FROM 'xxxbarxxx');

bar

>SELECT TRIM(TRAILING 'xyz' FROM 'barxxyz');

barx

UCASE(str)

UCASE() is a synonym for UPPER().

UNHEX(str)

Performs the inverse operation of HEX(str). That is, it interprets each pair of hexadecimal digits in the argument as a number and converts it to the character represented by the number. The resulting characters are returned as a binary string.

>SELECT UNHEX('4D7955314C');

MyUIL

UPPER(str)

Returns the string str with all characters changed to uppercase according to the current character set mapping.

>SELECT UPPER('Allah-hus-samad');

ALLAH-HUS-SAMAD

# SQL - Date Functions

1.ADDDATE(date,INTERVAL expr unit), ADDDATE(expr,days)

>SELECT DATE\_ADD('1998-01-02', INTERVAL 31 DAY);

1998-02-02

>SELECT ADDDATE('1998-01-02', 31);

1998-02-02

ADDTIME(expr1,expr2)

ADDTIME() adds expr2 to expr1 and returns the result. expr1 is a time or datetime expression, and expr2 is a time expression.

>SELECT ADDTIME('1997-12-31 23:59:59.999999','1 1:1:1.000002');

1998-01-02  01:01:01.000001

CONVERT\_TZ(dt,from\_tz,to\_tz)

This converts a datetime value dt from the time zone given by from\_tz to the time zone given by to\_tz and returns the resulting value. This function returns NULL if the arguments are invalid.

>SELECT CONVERT\_TZ('2004-01-01 12:00:00','+00:00','+10:00');

2004-01-01 22:00:00

CURDATE()

Returns the current date as a value in 'YYYY-MM-DD' or YYYYMMDD format, depending on whether the function is used in a string or numeric context.

>SELECT CURDATE();

2017-03-08

>SELECT CURDATE() + 0;

20170308

CURRENT\_DATE and CURRENT\_DATE()

CURRENT\_DATE and CURRENT\_DATE() are synonyms for CURDATE()

CURTIME()

Returns the current time as a value in 'HH:MM:SS' or HHMMSS format, depending on whether the function is used in a string or numeric context. The value is expressed in the current time zone.

>SELECT CURTIME();

11:07:45

>SELECT CURTIME() + 0;

110745

CURRENT\_TIME and CURRENT\_TIME()

CURRENT\_TIME and CURRENT\_TIME() are synonyms for CURTIME().

CURRENT\_TIMESTAMP and CURRENT\_TIMESTAMP()

CURRENT\_TIMESTAMP and CURRENT\_TIMESTAMP() are synonyms for NOW().

DATE(expr)

Extracts the date part of the date or datetime expression expr.

>SELECT DATE('2003-12-31 01:02:03');

2003-12-31

DATEDIFF(expr1,expr2)

DATEDIFF() returns expr1 . expr2 expressed as a value in days from one date to the other. expr1 and expr2 are date or date-and-time expressions. Only the date parts of the values are used in the calculation.

>SELECT DATEDIFF('1997-12-31 23:59:59','1997-12-30');

1

DATE\_FORMAT(date,format)

Formats the date value according to the format string.

>SELECT DATE\_FORMAT('1997-10-04 22:23:00', '%W %M %Y');

Saturday October 1997

Ex:

|  |  |
| --- | --- |
| %W | Weekday name (Sunday..Saturday) |
| %M | Month name(jan..dec) |
| %Y | Year, numeric four digits |

DATE\_SUB(date,INTERVAL expr unit)

This is similar to DATE\_ADD() function.

DAY(date)

DAY() is a synonym for DAYOFMONTH().

DAYNAME(date)

Returns the name of the weekday for date.

>SELECT DAYNAME('1998-02-05');

Thursday

DAYOFWEEK(date)

Returns the weekday index for date (1 = Sunday, 2 = Monday, ., 7 = Saturday). These index values correspond to the ODBC standard.

>SELECT DAYOFWEEK('1998-02-03');

3

DAYOFYEAR(date)

Returns the day of the year for date, in the range 1 to 366.

>SELECT DAYOFYEAR('1998-02-03');

34

EXTRACT(unit FROM date)

The EXTRACT() function uses the same kinds of unit specifiers as DATE\_ADD() or DATE\_SUB(), but extracts parts from the date rather than performing date arithmetic.

>SELECT EXTRACT(YEAR FROM '1999-07-02');

1999

>SELECT EXTRACT(YEAR\_MONTH FROM '1999-07-02 01:02:03');

199907

FROM\_DAYS(N)

Given a day number N, returns a DATE value.

>SELECT FROM\_DAYS(729669);

1997-10-07

FROM\_UNIXTIME(unix\_timestamp)

FROM\_UNIXTIME(unix\_timestamp,format)

Returns a representation of the unix\_timestamp argument as a value in 'YYYY-MM-DD HH:MM:SS' or YYYYMMDDHHMMSS format, depending on whether the function is used in a string or numeric context.

>SELECT FROM\_UNIXTIME(875996580);

1997-10-04 22:33:00

HOUR(time)

Returns the hour for time. The range of the return value is 0 to 23 for time-of-day values.

>SELECT HOUR('10:05:03');

10

LAST\_DAY(date)

Takes a date or datetime value and returns the corresponding value for the last day of the month. Returns NULL if the argument is invalid.

>SELECT LAST\_DAY('2003-02-05');

2003-02-28

LOCALTIME and LOCALTIME()

LOCALTIME and LOCALTIME() are synonyms for NOW().

LOCALTIMESTAMP and LOCALTIMESTAMP()

LOCALTIMESTAMP and LOCALTIMESTAMP() are synonyms for NOW().

MAKEDATE(year,dayofyear)

Returns a date, given year and day-of-year values. dayofyear must be greater than 0 or the result is NULL.

>SELECT MAKEDATE(2001,31), MAKEDATE(2001,32);

2001-01-31  2002-02-01

MAKETIME(hour,minute,second)

Returns a time value calculated from the hour, minute, and second arguments.

>SELECT MAKETIME(12,15,30);

12:15:30

MICROSECOND(expr)

Returns the microseconds from the time or datetime expression expr as a number in the range from 0 to 999999.

>SELECT MICROSECOND('12:00:00.123456');

123456

MINUTE(time)

Returns the minute for time, in the range 0 to 59.

>SELECT MINUTE('98-02-03 10:05:03');

5

MONTH(date)

Returns the month for date, in the range 0 to 12.

>SELECT MONTH('1998-02-03')

2

MONTHNAME(date)

Returns the full name of the month for date.

>SELECT MONTHNAME('1998-02-05');

February

NOW()

Returns the current date and time as a value in 'YYYY-MM-DD HH:MM:SS' or YYYYMMDDHHMMSS format, depending on whether the function is used in a string or numeric context. The value is expressed in the current time zone.

>SELECT NOW();

2017-03-09 2:31:45

PERIOD\_ADD(P,N)

Adds N months to period P (in the format YYMM or YYYYMM). Returns a value in the format YYYYMM. Note that the period argument P is not a date value.

>SELECT PERIOD\_ADD(9801,2);

199803

PERIOD\_DIFF(P1,P2)

Returns the number of months between periods P1 and P2. P1 and P2 should be in the format YYMM or YYYYMM. Note that the period arguments P1 and P2 are not date values.

>SELECT PERIOD\_DIFF(9802,199703);

11

QUARTER(date)

Returns the quarter of the year for date, in the range 1 to 4.

>SELECT QUARTER('98-04-01');

2

SECOND(time)

Returns the second for time, in the range 0 to 59.

>SELECT SECOND('10:05:03');

3

SEC\_TO\_TIME(seconds)

Returns the seconds argument, converted to hours, minutes and seconds, as a value in 'HH:MM:SS' or HHMMSS format, depending on whether the function is used in a string or numeric context.

>SELECT SEC\_TO\_TIME(2378);

00:39:38

STR\_TO\_DATE(str,format)

This is the inverse of the DATE\_FORMAT() function. It takes a string str and a format string format. STR\_TO\_DATE() returns a DATETIME value if the format string contains both date and time parts or a DATE or TIME value if the string contains only date or time parts.

>SELECT STR\_TO\_DATE('04/31/2004', '%m/%d/%Y');

2004-04-31

SUBDATE(date,INTERVAL expr unit) and SUBDATE(expr,days)

When invoked with the INTERVAL form of the second argument, SUBDATE() is a synonym for DATE\_SUB(). For information on the INTERVAL unit argument, see the discussion for DATE\_ADD().

>SELECT DATE\_SUB('1998-01-02', INTERVAL 31 DAY);

1998-01-02

SUBTIME(expr1,expr2)

SUBTIME() returns expr1 . expr2 expressed as a value in the same format as expr1. expr1 is a time or datetime expression, and expr2 is a time.

>SELECT SUBTIME('1997-12-31 23:59:59.999999',

    '1 1:1:1.000002');

1997-12-30 22:58:58.999997

SYSDATE()

Returns the current date and time as a value in 'YYYY-MM-DD HH:MM:SS' or YYYYMMDDHHMMSS format, depending on whether the function is used in a string or numeric context.

>SELECT SYSDATE();

2017-03-09 15:03:03

TIME(expr)

Extracts the time part of the time or datetime expression expr and returns it as a string.

>SELECT TIME('2003-12-31 01:02:03');

01:02:03

TIMEDIFF(expr1,expr2)

TIMEDIFF() returns expr1 . expr2 expressed as a time value. expr1 and expr2 are time or date-and-time expressions, but both must be of the same type.

>SELECT TIMEDIFF('1997-12-31 23:59:59.000001',

    '1997-12-30 01:01:01.000002');

46:58:57.999999

TIMESTAMP(expr), TIMESTAMP(expr1,expr2)

With a single argument, this function returns the date or datetime expression expr as a datetime value. With two arguments, it adds the time expression expr2 to the date or datetime expression expr1 and returns the result as a datetime value.

>SELECT TIMESTAMP('2003-12-31');

2003-12-31 00:00:00

TIMESTAMPADD(unit,interval,datetime\_expr)

Adds the integer expression interval to the date or datetime expression datetime\_expr.

>SELECT TIMESTAMPADD(MINUTE,1,'2003-01-02');

2003:01:02 00:01:00

TIMESTAMPDIFF(unit,datetime\_expr1,datetime\_expr2)

Returns the integer difference between the date or datetime expressions datetime\_expr1 and datetime\_expr2.

>SELECT TIMESTAMPDIFF(MONTH,'2003-02-01','2003-05-01');

3

TIME\_FORMAT(time,format)

This is used like the DATE\_FORMAT() function, but the format string may contain format specifiers only for hours, minutes and seconds.

>SELECT TIME\_FORMAT('100:00:00', '%H %k %h %I %l');

100 100 04 04 4

TIME\_TO\_SEC(time)

Returns the time argument converted to seconds.

>SELECT TIME\_TO\_SEC('22:23:00');

80580

TO\_DAYS(date)

Given a date, returns a day number (the number of days since year 0).

>SELECT TO\_DAYS(950501);

728779

UNIX\_TIMESTAMP(), UNIX\_TIMESTAMP(date)

If called with no argument, returns a Unix timestamp (seconds since '1970-01-01 00:00:00' UTC) as an unsigned integer.

>SELECT UNIX\_TIMESTAMP();

8822226357

>SELECT UNIX\_TIMESTAMP('1997-10-04 22:23:00');

875996580

UTC\_DATE, UTC\_DATE()

Returns the current UTC date as a value in 'YYYY-MM-DD' or YYYYMMDD format, depending on whether the function is used in a string or numeric context.

>SELECT UTC\_DATE(), UTC\_DATE() + 0;

2017-03-09        20170309

UTC\_TIME, UTC\_TIME()

Returns the current UTC time as a value in 'HH:MM:SS' or HHMMSS format, depending on whether the function is used in a string or numeric context.

>SELECT UTC\_TIME(), UTC\_TIME() + 0;

03:58:12     035812

UTC\_TIMESTAMP, UTC\_TIMESTAMP()

Returns the current UTC date and time as a value in 'YYYY-MM-DD HH:MM:SS' or YYYYMMDDHHMMSS format, depending on whether the function is used in a string or numeric context.

>SELECT UTC\_TIMESTAMP(), UTC\_TIMESTAMP() + 0;

2017-03-09 04:01:45   20170309040145

WEEK(date[,mode])

This function returns the week number for date.

>SELECT WEEK('1998-02-20');

7

WEEKDAY(date)

Returns the weekday index for date (0 = Monday, 1 = Tuesday, . 6 = Sunday).

>SELECT WEEKDAY('1998-02-03 22:23:00');

1

WEEKOFYEAR(date)

Returns the calendar week of the date as a number in the range from 1 to 53. WEEKOFYEAR() is a compatibility function that is equivalent to WEEK(date,3).

>SELECT WEEKOFYEAR('1998-02-20');

8

YEAR(date)

Returns the year for date, in the range 1000 to 9999, or 0 for the .zero. date.

>SELECT YEAR('98-02-03');

1998

YEARWEEK(date), YEARWEEK(date,mode)

Returns year and week for a date. The mode argument works exactly like the mode argument to WEEK(). The year in the result may be different from the year in the date argument for the first and the last week of the year.

>SELECT YEARWEEK('1987-01-01');

198652