# SQL ─ Overview

SQL is a language to operate databases; it includes database creation, deletion, fetching rows, modifying rows, etc. SQL is an **ANSI** (American National Standards Institute) standard language, but there are many different versions of the SQL language.

## What is SQL?

SQL is Structured Query Language, which is a computer language for storing, manipulating and retrieving data stored in a relational database.

SQL is the standard language for Relational Database System. All the Relational Database Management Systems (RDMS) like MySQL, MS Access, Oracle, Sybase, Informix, Postgres and SQL Server use SQL as their standard database language.

Also, they are using different dialects, such as:

* MS SQL Server using T-SQL,
* Oracle using PL/SQL,
* MS Access version of SQL is called JET SQL (native format) etc.

### Why SQL?

SQL is widely popular because it offers the following advantages:

* Allows users to access data in the relational database management systems.
* Allows users to describe the data.
* Allows users to define the data in a database and manipulate that data.
* Allows to embed within other languages using SQL modules, libraries & pre-compilers.
* Allows users to create and drop databases and tables.
* Allows users to create view, stored procedure, functions in a database.
* Allows users to set permissions on tables, procedures and views.

### A Brief History of SQL

* 1970 – Dr. Edgar F. "Ted" Codd of IBM is known as the father of relational databases. He described a relational model for databases.
* 1974 – Structured Query Language appeared.
* 1978 – IBM worked to develop Codd's ideas and released a product named System/R.
* 1986 – IBM developed the first prototype of relational database and standardized by ANSI. The first relational database was released by Relational Software which later came to be known as Oracle.

## 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 this process.

These components are –

* Query Dispatcher
* Optimization Engines
* Classic Query Engine
* SQL Query Engine, etc.

A classic query engine handles all the non-SQL queries, but a SQL query engine won't handle logical files.

Following is a simple diagram showing the SQL Architecture:

## SQL Commands

The standard SQL commands to interact with relational databases are CREATE, SELECT, INSERT, UPDATE, DELETE and DROP. These commands can be classified into the following groups based on their nature:

**DDL - Data Definition Language**

|  |  |
| --- | --- |
| **Command** | **Description** |
| CREATE | Creates a new table, a view of a table, or other object in the database. |
| ALTER | Modifies an existing database object, such as a table. |
| DROP | Deletes an entire table, a view of a table or other objects in the database. |

**DML - Data Manipulation Language**

|  |  |
| --- | --- |
| **Command** | **Description** |
| SELECT | Retrieves certain records from one or more tables. |
| INSERT | Creates a record. |
| UPDATE | Modifies records. |
| DELETE | Deletes records. |

**DCL - Data Control Language**

|  |  |
| --- | --- |
| **Command** | **Description** |
| GRANT | Gives a privilege to user. |
| EVOKE | Takes back privileges granted from user. |

# SQL ─ RDBMS Concepts

## What is RDBMS?

RDBMS stands for **R**elational **D**atabase **M**anagement **S**ystem. RDBMS is the basis for SQL, and for all modern database systems like MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.

A Relational database management system (RDBMS) is a database management system (DBMS) that is based on the relational model as introduced by E. F. Codd.

### What is a table?

The data in an RDBMS is stored in database objects which are called as **tables**. This table is basically a collection of related data entries and it consists of numerous columns and rows.

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

| 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 |

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

Remember, a table is the most common and simplest form of data storage in a relational database. The following program is an example of a CUSTOMERS table:

### What is a field?

Every table is broken up into smaller entities called fields. The fields in the CUSTOMERS table consist of ID, NAME, AGE, ADDRESS and SALARY.

A field is a column in a table that is designed to maintain specific information about every record in the table.

### What is a Record or a Row?

A record is also called as a row of data is each individual entry that exists in a table. For example, there are 7 records in the above CUSTOMERS table. Following is a single row of data or record in the CUSTOMERS table:

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

| 1 | Ramesh   | 32 | Ahmedabad |  2000.00 |

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

A record is a horizontal entity in a table.

### What is a column?

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

For example, a column in the CUSTOMERS table is ADDRESS, which represents location description and would be as shown below:

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

| ADDRESS  |

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

| Ahmedabad |

| Delhi  |

| Kota   |

| Mumbai   |

| Bhopal   |

| MP   |

| Indore   |

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

### What is a NULL value?

A NULL value in a table is a value in a field that appears to be blank, which means a field with a NULL value is a field with no value.

It is very important to understand that a NULL value is different than a zero value or a field that contains spaces. A field with a NULL value is the one that has been left blank during a record creation.

## SQL Constraints

Constraints are the rules enforced on data columns on a 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 can either be column level or table level. Column level constraints are applied only to one column whereas, table level constraints are applied to the entire table.

Following are some of the most commonly used constraints available in SQL:

* NOT NULL Constraint: Ensures that a column cannot have a NULL value.
* DEFAULT Constraint: Provides a default value for a column when none is specified.
* UNIQUE Constraint: Ensures that all the values in a column are different.
* PRIMARY Key: Uniquely identifies each row/record in a database table.
* FOREIGN Key: Uniquely identifies a row/record in any another database table.
* CHECK Constraint: The CHECK constraint ensures that all values in a column satisfy certain conditions.
* INDEX: Used to create and retrieve data from the database very quickly.

## Data Integrity

* The following categories of data integrity exist with each RDBMS:
* Entity Integrity: There are no duplicate rows in a table.
* Domain Integrity: Enforces valid entries for a given column by restricting the type, the format, or the range of values.
* Referential integrity: Rows cannot be deleted, which are used by other records.
* User-Defined Integrity: Enforces some specific business rules that do not fall into entity, domain or referential integrity.

## Database Normalization

Database normalization is the process of efficiently organizing data in a database. There are two reasons of this normalization process:

* Eliminating redundant data. For example, storing the same data in more than one table.
* Ensuring data dependencies make sense.

Both these reasons are worthy goals as they reduce the amount of space a database consumes and ensures that data is logically stored. Normalization consists of a series of guidelines that help guide you in creating a good database structure.

Normalization guidelines are divided into normal forms; think of a form as the format or the way a database structure is laid out. The aim of normal forms is to organize the database structure, so that it complies with the rules of first normal form, then second normal form and finally the third normal form.

It is your choice to take it further and go to the fourth normal form, fifth normal form and so on, but in general, the third normal form is more than enough.

* First Normal Form (1NF)
* Second Normal Form (2NF)
* Third Normal Form (3NF)

## Database – First Normal Form (1NF)

The First normal form (1NF) sets basic rules for an organized database:

* Define the data items required, because they become the columns in a table.
* Place the related data items in a table.
* Ensure that there are no repeating groups of data.
* Ensure that there is a primary key.

### First Rule of 1NF

You must define the data items. This means looking at the data to be stored, organizing the data into columns, defining what type of data each column contains and then finally putting the related columns into their own table.

For example, you put all the columns relating to locations of meetings in the Location table, those relating to members in the MemberDetails table and so on.

### Second Rule of 1NF

The next step is ensuring that there are no repeating groups of data. Consider we have the following table:

CREATE TABLE CUSTOMERS(

ID  INT NOT NULL,

NAME VARCHAR (20) AGE INT NOT NULL,

ADDRESS CHAR (25),

ORDERS  VARCHAR(155)

);

So, if we populate this table for a single customer having multiple orders, then it would be something as shown below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **ORDERS** |
| 100 | Sachin | 36 | Lower West Side | Cannon XL-200 |
| 100 | Sachin | 36 | Lower West Side | Battery XL-200 |
| 100 | Sachin | 36 | Lower West Side | Tripod Large |

But as per the 1NF, we need to ensure that there are no repeating groups of data. So, let us break the above table into two parts and then join them using a key as shown in the following program:

**CUSTOMERS Table**

CREATE TABLE CUSTOMERS(

ID  INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25),

PRIMARY KEY (ID)

);

This table would have the following record:

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** |
| 100 | Sachin | 36 | Lower West Side |

**ORDERS Table**

CREATE TABLE ORDERS(

ID  INT NOT NULL,

CUSTOMER\_ID INT NOT NULL,

ORDERS  VARCHAR(155),

PRIMARY KEY (ID)

);

This table would have the following records:

|  |  |  |
| --- | --- | --- |
| **ID** | **CUSTOMER\_ID** | **ORDERS** |
| 10 | 100 | Cannon XL-200 |
| 11 | 100 | Battery XL-200 |
| 12 | 100 | Tripod Large |

**Third Rule of 1NF**

The final rule of the first normal form, create a primary key for each table which we have already created.

## Database – Second Normal Form (2NF)

The Second Normal Form states that it should meet all the rules for 1NF and there must be no partial dependences of any of the columns on the primary key:

Consider a customer-order relation and you want to store customer ID, customer name, order ID and order detail and the date of purchase:

CREATE TABLE CUSTOMERS(

CUST\_ID INT NOT NULL,

CUST\_NAME VARCHAR (20) NOT NULL,

ORDER\_ID NOT NULL,

ORDER\_DETAIL VARCHAR (20) NOT NULL,

SALE\_DATE DATETIME,

PRIMARY KEY (CUST\_ID, ORDER\_ID)

);

This table is in the first normal form; in that it obeys all the rules of the first normal form. In this table, the primary key consists of the CUST\_ID and the ORDER\_ID. Combined, they are unique assuming the same customer would hardly order the same thing.

However, the table is not in the second normal form because there are partial dependencies of primary keys and columns. CUST\_NAME is dependent on CUST\_ID and there's no real link between a customer's name and what he purchased. The order detail and purchase date are also dependent on the ORDER\_ID, but they are not dependent on the CUST\_ID, because there is no link between a CUST\_ID and an ORDER\_DETAIL or their SALE\_DATE.

To make this table comply with the second normal form, you need to separate the columns into three tables.

First, create a table to store the customer details as shown in the code block below:

CREATE TABLE CUSTOMERS(

CUST\_ID INT NOT NULL,

CUST\_NAME VARCHAR (20) NOT NULL,

PRIMARY KEY (CUST\_ID)

);

The next step is to create a table to store the details of each order:

CREATE TABLE ORDERS(

ORDER\_ID  INT NOT NULL,

ORDER\_DETAIL VARCHAR (20) NOT NULL,

PRIMARY KEY (ORDER\_ID)

);

Finally, create a third table storing just the CUST\_ID and the ORDER\_ID to keep a track of all the orders for a customer:

CREATE TABLE CUSTMERORDERS(

CUST\_ID INT NOT NULL,

ORDER\_ID INT NOT NULL,

SALE\_DATE DATETIME,

PRIMARY KEY (CUST\_ID, ORDER\_ID)

);

## Database – Third Normal Form (3NF)

A table is in a third normal form when the following conditions are met:

* It is in the second normal form.
* All non-primary fields are dependent on the primary key.

The dependency of these non-primary fields is between the data. For example, in the following table – the street name, city and the state are unbreakably bound to their zip code.

CREATE TABLE CUSTOMERS(

CUST\_ID INT NOT NULL,

CUST\_NAME VARCHAR (20) NOT NULL,

DOB DATE,

STREET VARCHAR(200),

CITY VARCHAR(100),

STATE VARCHAR(100),

ZIP VARCHAR(12),

EMAIL\_ID VARCHAR(256),

PRIMARY KEY (CUST\_ID)

);

The dependency between the zip code and the address is called as a transitive dependency. To comply with the third normal form, all you need to do is to move the Street, City and the State fields into their own table, which you can call as the Zip Code table.

CREATE TABLE ADDRESS(

ZIP  VARCHAR(12),

STREET  VARCHAR(200),

CITY   VARCHAR(100),

STATE  VARCHAR(100), PRIMARY KEY (ZIP)

)

The next step is to alter the CUSTOMERS table as shown below.

CREATE TABLE CUSTOMERS(

CUST\_ID INT NOT NULL,

CUST\_NAME VARCHAR (20) NOT NULL,

DOB DATE,

ZIP VARCHAR(12),

EMAIL\_ID VARCHAR(256),

PRIMARY KEY (CUST\_ID)

);

The advantages of removing transitive dependencies are mainly two-fold. First, the amount of data duplication is reduced and therefore your database becomes smaller.

The second advantage is data integrity. When duplicated data changes, there is a big risk of updating only some of the data, especially if it is spread out in many different places in the database.

For example, if the address and the zip code data were stored in three or four different tables, then any changes in the zip codes would need to ripple out to every record in those three or four tables.

# SQL ─ RDBMS Databases

There are many popular RDBMS available to work with. This tutorial gives a brief overview of some of the most popular RDBMS’s. This would help you to compare their basic features.

## MySQL

MySQL is an open source SQL database, which is developed by a Swedish company – MySQL AB. MySQL is pronounced as "my ess-que-ell," in contrast with SQL, pronounced "sequel."

MySQL is supporting many different platforms including Microsoft Windows, the major Linux distributions, UNIX, and Mac OS X.

MySQL has free and paid versions, depending on its usage (non-commercial/commercial) and features. MySQL comes with a very fast, multi-threaded, multi-user and robust SQL database server.

### History

* Development of MySQL by Michael Widenius & David Axmark beginning in 1994.
* First internal release on 23rd May 1995.
* Windows Version was released on the 8th January 1998 for Windows 95 and NT.
* Version 3.23: beta from June 2000, production release January 2001.
* Version 4.0: beta from August 2002, production release March 2003 (unions).
* Version 4.01: beta from August 2003, Jyoti adopts MySQL for database tracking.
* Version 4.1: beta from June 2004, production release October 2004.
* Version 5.0: beta from March 2005, production release October 2005.
* Sun Microsystems acquired MySQL AB on the 26th February 2008.
* Version 5.1: production release 27th November 2008.

### Features

* High Performance.
* High Availability.
* Scalability and Flexibility Run anything.
* Robust Transactional Support.
* Web and Data Warehouse Strengths.
* Strong Data Protection.
* Comprehensive Application Development.
* Management Ease.
* Open Source Freedom and 24 x 7 Support.
* Lowest Total Cost of Ownership.

## MS SQL Server

MS SQL Server is a Relational Database Management System developed by Microsoft Inc. Its primary query languages are:

* T-SQL
* ANSI SQL

### History

* 1987 - Sybase releases SQL Server for UNIX.
* 1988 - Microsoft, Sybase, and Aston-Tate port SQL Server to OS/2.
* 1989 - Microsoft, Sybase, and Aston-Tate release SQL Server 1.0 for OS/2.
* 1990 - SQL Server 1.1 is released with support for Windows 3.0 clients.
* Aston - Tate drops out of SQL Server development.
* 2000 - Microsoft releases SQL Server 2000.
* 2001 - Microsoft releases XML for SQL Server Web Release 1 (download).
* 2002 - Microsoft releases SQLXML 2.0 (renamed from XML for SQL Server).
* 2002 - Microsoft releases SQLXML 3.0.
* 2005 - Microsoft releases SQL Server 2005 on November 7th, 2005.

### Features

* High Performance
* High Availability
* Database mirroring
* Database snapshots
* CLR integration
* Service Broker
* DDL triggers
* Ranking functions
* Row version-based isolation levels
* XML integration
* TRY...CATCH
* Database Mail

## ORACLE

It is a very large multi-user based database management system. Oracle is a relational database management system developed by 'Oracle Corporation'.

Oracle works to efficiently manage its resources, a database of information among the multiple clients requesting and sending data in the network.

It is an excellent database server choice for client/server computing. Oracle supports all major operating systems for both clients and servers, including MSDOS, NetWare, UnixWare, OS/2 and most UNIX flavors.

### History

Oracle began in 1977 and celebrating its 32 wonderful years in the industry (from 1977 to 2009).

* 1977 - Larry Ellison, Bob Miner and Ed Oates founded Software Development Laboratories to undertake development work.
* 1979 - Version 2.0 of Oracle was released and it became first commercial relational database and first SQL database. The company changed its name to Relational Software Inc. (RSI).
* 1981 - RSI started developing tools for Oracle.
* 1982 - RSI was renamed to Oracle Corporation.
* 1983 - Oracle released version 3.0, rewritten in C language and ran on multiple platforms.
* 1984 - Oracle version 4.0 was released. It contained features like concurrency control - multi-version read consistency, etc.
* 1985 - Oracle version 4.0 was released. It contained features like concurrency control - multi-version read consistency, etc.
* 2007 - Oracle released Oracle11g. The new version focused on better partitioning, easy migration, etc.

### Features

* Concurrency
* Read Consistency
* Locking Mechanisms
* Quiesce Database
* Portability
* Self-managing database
* SQL\*Plus
* ASM
* Scheduler
* Resource Manager
* Data Warehousing
* Materialized views
* Bitmap indexes
* Table compression
* Parallel Execution
* Analytic SQL
* Data mining
* Partitioning

## MS ACCESS

This is one of the most popular Microsoft products. Microsoft Access is an entry-level database management software. MS Access database is not only inexpensive but also a powerful database for small-scale projects.

MS Access uses the Jet database engine, which utilizes a specific SQL language dialect (sometimes referred to as Jet SQL).

MS Access comes with the professional edition of MS Office package. MS Access has easy- to-use intuitive graphical interface.

* 1992 - Access version 1.0 was released.
* 1993 - Access 1.1 released to improve compatibility with inclusion the Access Basic programming language.
* The most significant transition was from Access 97 to Access 2000
* 2007 - Access 2007, a new database format was introduced ACCDB which supports complex data types such as multi valued and attachment fields.

### Features

* Users can create tables, queries, forms and reports and connect them together with macros.
* Option of importing and exporting the data to many formats including Excel, Outlook, ASCII, dBase, Paradox, FoxPro, SQL Server, Oracle, ODBC, etc.
* There is also the Jet Database format (MDB or ACCDB in Access 2007), which can contain the application and data in one file. This makes it very convenient to distribute the entire application to another user, who can run it in disconnected environments.
* Microsoft Access offers parameterized queries. These queries and Access tables can be referenced from other programs like VB6 and .NET through DAO or ADO.
* The desktop editions of Microsoft SQL Server can be used with Access as an alternative to the Jet Database Engine.
* Microsoft Access is a file server-based database. Unlike the client-server relational database management systems (RDBMS), Microsoft Access does not implement database triggers, stored procedures or transaction logging.

# SQL – Syntax

SQL is followed by a unique set of rules and guidelines called Syntax. This tutorial gives you a quick start with SQL by listing all the basic SQL Syntax.

All the SQL statements start with any of the keywords like SELECT, INSERT, UPDATE, DELETE, ALTER, DROP, CREATE, USE, SHOW and all the statements end with a semicolon (;).

The most important point to be noted here is that SQL is **case insensitive**, which means SELECT and select have same meaning in SQL statements. Whereas, MySQL makes difference in table names. So, if you are working with MySQL, then you need to give table names as they exist in the database.

## Various Syntax in SQL

All the examples given in this tutorial have been tested with a MySQL server.

**SQL SELECT Statement**

SELECT column1, column2....columnN

FROM table\_name;

**SQL DISTINCT Clause**

SELECT DISTINCT column1, column2....columnN

FROM  table\_name;

**SQL WHERE Clause**

SELECT column1, column2....columnN

FROM  table\_name

WHERE CONDITION;

**SQL AND/OR Clause**

SELECT column1, column2....columnN

FROM  table\_name

WHERE CONDITION-1 {AND|OR} CONDITION-2;

**SQL IN Clause**

SELECT column1, column2....columnN

FROM  table\_name

WHERE column\_name IN (val-1, val-2,...val-N);

**SQL BETWEEN Clause**

SELECT column1, column2....columnN

FROM  table\_name

WHERE column\_name BETWEEN val-1 AND val-2;

**SQL LIKE Clause**

SELECT column1, column2....columnN

FROM  table\_name

WHERE column\_name LIKE { PATTERN };

**SQL ORDER BY Clause**

SELECT column1, column2....columnN

FROM  table\_name

WHERE CONDITION

ORDER BY column\_name {ASC|DESC};

**SQL GROUP BY Clause**

SELECT SUM(column\_name)

FROM  table\_name

WHERE CONDITION

GROUP BY column\_name;

**SQL COUNT Clause**

SELECT COUNT(column\_name)

FROM  table\_name

WHERE CONDITION;

**SQL HAVING Clause**

SELECT SUM(column\_name)

FROM  table\_name

WHERE CONDITION

GROUP BY column\_name

HAVING (arithematic function condition);

**SQL CREATE TABLE Statement**

CREATE TABLE table\_name(

column1 datatype,

column2 datatype,

column3 datatype,

.....

columnN datatype,

PRIMARY KEY( one or more columns )

);

**SQL DROP TABLE Statement**

DROP TABLE table\_name;

**SQL CREATE INDEX Statement**

CREATE UNIQUE INDEX index\_name

ON table\_name ( column1, column2,...columnN);

**SQL DROP INDEX Statement**

ALTER TABLE table\_name

DROP INDEX index\_name;

**SQL DESC Statement**

DESC table\_name;

**SQL TRUNCATE TABLE Statement**

TRUNCATE TABLE table\_name;

**SQL ALTER TABLE Statement**

ALTER TABLE table\_name {ADD|DROP|MODIFY} column\_name {data\_ype};

**SQL ALTER TABLE Statement (Rename)**

ALTER TABLE table\_name RENAME TO new\_table\_name;

**SQL INSERT INTO Statement**

INSERT INTO table\_name( column1, column2....columnN)

VALUES ( value1, value2....valueN);

**SQL UPDATE Statement**

UPDATE table\_name

SET column1 = value1, column2 =

value2....columnN=valueN [ WHERE CONDITION ];

**SQL DELETE Statement**

DELETE FROM table\_name

WHERE {CONDITION};

**SQL CREATE DATABASE Statement**

CREATE DATABASE database\_name;

**SQL DROP DATABASE Statement**

DROP DATABASE database\_name;

**SQL USE Statement**

USE database\_name;

**SQL COMMIT Statement**

COMMIT;

**SQL ROLLBACK Statement**

ROLLBACK;

# SQL ─ Data Types

SQL Data Type is an attribute that specifies the type of data of any object. Each column, variable and expression has a related data type in SQL. You can use these data types while creating your tables. You can choose a data type for a table column based on your requirement.

SQL Server offers six categories of data types for your use which are listed below −

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

**Exact Numeric Data Types**

**Approximate Numeric Data Types**

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

**Date and Time Data Types**

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

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

**Character Strings Data Types**

|  |  |
| --- | --- |
| **DATA TYPE** | **Description** |
| **char** | **Maximum length of 8,000 characters.( Fixed length non- Unicode characters)** |
| **varchar** | **Maximum of 8,000 characters.(Variable-length non-Unicode data).** |
| **varchar(max)** | **Maximum length of 231characters, Variable-length non- Unicode data (SQL Server 2005 only).** |
| **text** | **Variable-length non-Unicode data with a maximum length of 2,147,483,647 characters.** |

**Unicode Character Strings Data Types**

|  |  |
| --- | --- |
| **DATA TYPE** | **Description** |
| **nchar** | **Maximum length of 4,000 characters.( Fixed length Unicode)** |
| **nvarchar** | **Maximum length of 4,000 characters.(Variable length Unicode)** |
| **nvarchar(max)** | **Maximum length of 231characters (SQL Server 2005 only).( Variable length Unicode)** |
| **ntext** | **Maximum length of 1,073,741,823 characters. ( Variable length Unicode )** |

**Binary Data Types**

|  |  |
| --- | --- |
| **DATA TYPE** | **Description** |
| **binary** | **Maximum length of 8,000 bytes(Fixed-length binary data )** |
| **varbinary** | **Maximum length of 8,000 bytes.(Variable length binary data)** |
| **varbinary(max)** | **Maximum length of 231 bytes (SQL Server 2005 only). ( Variable length Binary data)** |
| **image** | **Maximum length of 2,147,483,647 bytes. ( Variable length Binary Data)** |

**Misc. Data Types**

|  |  |
| --- | --- |
| **DATA TYPE** | **Description** |
| **sql\_variant** | **Stores values of various SQL Server-supported data types, except text, ntext, and timestamp.** |
| **timestamp** | **Stores a database-wide unique number that gets updated every time a row gets updated** |
| **uniqueidentifier** | **Stores a globally unique identifier (GUID)** |
| **xml** | **Stores XML data. You can store xml instances in a column or a variable (SQL Server 2005 only).** |
| **cursor** | **Reference to a cursor object** |
| **table** | **Stores a result set for later processing** |

# SQL – Operators

## What is an Operator in SQL?

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

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

## SQLArithmetic Operators

**Assume ‘variable a’ holds 10 and ‘variable b’ holds 20, then:**

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

## Arithmetic Operators – Examples

Here are a few simple examples showing the usage of SQL Arithmetic Operators:

**Example 1:**

*SQL*> select 10+ 20;

**Output:**

+--------+

| 10+ 20 |

+--------+

| 30 |

+--------+

1 row in set (0.00 sec)

**Example 2:**

*SQL*> select 10 \* 20;

**Output:**

+---------+

| 10 \* 20 |

+---------+

| 200 |

+---------+

1 row in set (0.00 sec)

**Example 3:**

*SQL*> select 10 / 5;

**Output:**

+--------+

| 10 / 5 |

+--------+

| 2.0000 |

+--------+

1 row in set (0.03 sec)

**Example 4:**

*SQL*> select 12 %  5;

**Output:**

+---------+

| 12 %  5 |

+---------+

| 2 |

+---------+

1 row in set (0.00 sec)

## SQL Comparison Operators

Assume ‘variable a’ holds 10 and ‘variable b’ holds 20, then:

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

## Comparison Operators – Examples

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 are some simple examples showing the usage of SQL Comparison Operators:

**Example 1:**

*SQL*> SELECT \* FROM CUSTOMERS WHERE SALARY > 5000;

**Output:**

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

| ID | NAME  | AGE | ADDRESS  | SALARY  |

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

| 4 | Chaitali  | 25  | Mumbai   | 6500.00 |

| 5 | Hardik   | 27  | Bhopal   | 8500.00 |

| 7 | Muffy  | 24  | Indore   | 10000.00|

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

**Example 2:**

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

**Output:**

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

| ID | NAME  | AGE | ADDRESS  | SALARY  |

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

| 1 | Ramesh   | 32  | Ahmedabad | 2000.00 |

| 3 | kaushik  | 23  | Kota   | 2000.00 |

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

**Example 3:**

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

**Output:**

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

| ID | NAME  | AGE | ADDRESS  | SALARY  |

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

| 2 | Khilan   | 25  | Delhi  | 1500.00 |

| 4 | Chaitali  | 25  | Mumbai   | 6500.00 |

| 5 | Hardik   | 27  | Bhopal   | 8500.00 |

| 6 | Komal  | 22  | MP   | 4500.00 |

| 7 | Muffy  | 24  | Indore   | 10000.00|

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

**Example 4:**

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

**Output:**

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

| ID | NAME  | AGE | ADDRESS  | SALARY  |

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

| 2 | Khilan   | 25  | Delhi  | 1500.00 |

| 4 | Chaitali  | 25  | Mumbai   | 6500.00 |

| 5 | Hardik   | 27  | Bhopal   | 8500.00 |

| 6 | Komal  | 22  | MP   | 4500.00 |

| 7 | Muffy  | 24  | Indore   | 10000.00|

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

**Example 5:**

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

**Output:**

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

| ID | NAME  | AGE | ADDRESS  | SALARY  |

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

| 4 | Chaitali  | 25  | Mumbai   | 6500.00 |

| 5 | Hardik   | 27  | Bhopal   | 8500.00 |

| 7 | Muffy  | 24  | Indore   | 10000.00|

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

## SQL Logical Operators

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

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

## Logical Operators – Examples

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|

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

**Here are some simple examples showing usage of SQL Comparison Operators:**

**Example 1:**

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

**Output:**

| 4 | Chaitali  | 25  | Mumbai   | 6500.00 |

| 5 | Hardik   | 27  | Bhopal   | 8500.00 |

**Example 2:**

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

**Output:**

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

| ID | NAME  | AGE | ADDRESS  | SALARY  |

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

| 1 | Ramesh   | 32  | Ahmedabad | 2000.00 |

| 2 | Khilan   | 25  | Delhi  | 1500.00 |

| 4 | Chaitali  | 25  | Mumbai   | 6500.00 |

| 5 | Hardik   | 27  | Bhopal   | 8500.00 |

| 7 | Muffy  | 24  | Indore   | 10000.00|

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

**Example 3:**

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

**Output:**

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

| 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|

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

**Example 4:**

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

**Output:**

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

| ID | NAME  | AGE | ADDRESS  | SALARY  |

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

| 6 | Komal  | 22  | MP   | 4500.00 |

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

**Example 5:**

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

**Output:**

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

| ID | NAME  | AGE | ADDRESS  | SALARY  |

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

| 2 | Khilan   | 25  | Delhi  | 1500.00 |

| 4 | Chaitali  | 25  | Mumbai   | 6500.00 |

| 5 | Hardik   | 27  | Bhopal   | 8500.00 |

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

**Example 6:**

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

**Output:**

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

| ID | NAME  | AGE | ADDRESS  | SALARY  |

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

| 2 | Khilan   | 25  | Delhi  | 1500.00 |

| 4 | Chaitali  | 25  | Mumbai   | 6500.00 |

| 5 | Hardik   | 27  | Bhopal   | 8500.00 |

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

**Example 7:**

*SQL*> SELECT AGE FROM CUSTOMERS

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

**Output:**

+-----+

| AGE |

+-----+

| 32 |

| 25 |

| 23 |

| 25 |

| 27 |

| 22 |

| 24 |

+-----+

7 rows in set (0.02 sec)

**Example 8:**

*SQL*> SELECT \* FROM CUSTOMERS

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

**Output:**

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

| ID | NAME  | AGE | ADDRESS  | SALARY  |

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

| 1 | Ramesh   | 32  | Ahmedabad | 2000.00 |

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

**Example 9:**

*SQL*> SELECT \* FROM CUSTOMERS

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

**Output:**

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

| ID | NAME  | AGE | ADDRESS  | SALARY  |

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

| 1 | Ramesh   | 32  | Ahmedabad | 2000.00 |

| 2 | Khilan   | 25  | Delhi  | 1500.00 |

| 4 | Chaitali  | 25  | Mumbai   | 6500.00 |

| 5 | Hardik   | 27  | Bhopal   | 8500.00 |

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

# SQL – Expressions

An expression is a combination of one or more values, operators and SQL functions that evaluate to a value. These SQL EXPRESSIONs are like formulae and they are written in query language. You can also use them to query the database for a 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:**

* Boolean
* Numeric
* Date

Let us now discuss each of these in detail.

## Boolean Expressions

SQL Boolean Expressions fetch the data based on matching a 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)

**The following table is a simple example showing the usage of various 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)

## Numeric Expressions

These expressions are 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, the numerical\_expression is used for a mathematical expression or any formula. Following is a simple example showing the 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 the 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)

## Date Expressions

Date Expressions return the 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 shown below:

*SQL*>  SELECT  GETDATE();;

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

| GETDATE  |

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

| 2009-10-22 12:07:18.140 |

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

1 row in set (0.00 sec)

# SQL – Create Database

The SQL CREATE DATABASE statement is used to create a new SQL database.

**Syntax**

The basic syntax of this CREATE DATABASE statement is as follows:

CREATE DATABASE DatabaseName;

Always the database name should be unique within the RDBMS.

**Example**

If you want to create a new database <testDB>, then the CREATE DATABASE statement would be as shown below:

*SQL*> CREATE DATABASE testDB;

Make sure you have the admin privilege before creating any database. Once a database is created, you can check it in the list of databases as follows:

*SQL*> SHOW DATABASES;

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

| Database   |

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

| information\_schema |

| AMROOD   |

| TUTORIALSPOINT   |

| mysql  |

| orig   |

| test   |

| testDB   |

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

7 rows in set (0.00 sec)

# SQL – Drop Database

The SQL DROP DATABASE statement is used to drop an existing database in SQL schema.

**Syntax**

The basic syntax of DROP DATABASE statement is as follows:

DROP DATABASE DatabaseName;

Always the database name should be unique within the RDBMS.

**Example**

If you want to delete an existing database <testDB>, then the DROP DATABASE statement would be as shown below:

*SQL*> DROP DATABASE testDB;

**NOTE**: Be careful before using this operation because by deleting an existing database would result in loss ofcomplete information stored in the database.

Make sure you have the admin privilege before dropping any database. Once a database is dropped, you can check it in the list of the databases as shown below:

*SQL*> SHOW DATABASES;

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

| Database   |

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

| information\_schema |

| AMROOD    |

| TUTORIALSPOINT   |

| mysql  |

| orig   |

| test   |

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

6 rows in set (0.00 sec)

# SQL ─ SELECT Database, USE Statement

When you have multiple databases in your SQL Schema, then before starting your operation, you would need to select a database where all the operations would be performed.

The SQL USE statement is used to select any existing database in the SQL schema.

**Syntax**

The basic syntax of the USE statement is as shown below:

USE DatabaseName;

Always the database name should be unique within the RDBMS.

**Example**

You can check the available databases as shown below:

*SQL*> SHOW DATABASES;

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

| Database   |

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

| information\_schema |

| AMROOD   |

| TUTORIALSPOINT   |

| mysql  |

| orig   |

| test   |

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

6 rows in set (0.00 sec)

Now, if you want to work with the AMROOD database, then you can execute the following SQL command and start working with the AMROOD database.

*SQL*> USE AMROOD;

# SQL ─ CREATE Table

Creating a basic table involves naming the table and defining its columns and each column's data type.

The SQL CREATE TABLE statement is used to create a new table.

**Syntax**

The basic syntax of the CREATE TABLE statement is as follows:

CREATE TABLE table\_name(

column1 datatype,

column2 datatype,

column3 datatype,

.....

columnN datatype,

PRIMARY KEY( one or more columns )

);

CREATE TABLE is the keyword telling the database system what you want to do. In this case, you want to create a new table. The unique name or identifier for the table follows the CREATE TABLE statement.

Then in brackets comes the list defining each column in the table and what sort of data type it is. The syntax becomes clearer with the following example.

A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement. You can check the complete details at Create Table Using another Table.

**Example**

The following code block is an example, which creates a CUSTOMERS table with an ID as a primary key and NOT NULL are the constraints showing that these fields cannot be NULL while creating records in this table:

*SQL*> CREATE TABLE CUSTOMERS(

ID  INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25) ,

SALARY  DECIMAL (18, 2), PRIMARY KEY (ID)

);

You can verify if your table has been created successfully by looking at the message displayed by the SQL server, otherwise you can use the DESC command as follows:

*SQL*> DESC CUSTOMERS;

Now, you have CUSTOMERS table available in your database which you can use to store the required information related to customers.

## SQL - Creating a Table from an Existing Table

A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement. The new table has the same column definitions. All columns or specific columns can be selected. When you will create a new table using the existing table, the new table would be populated using the existing values in the old table.

**Syntax**

The basic syntax for creating a table from another table is as follows:

CREATE TABLE NEW\_TABLE\_NAME AS

SELECT [ column1, column2...columnN ]

FROM EXISTING\_TABLE\_NAME

[ WHERE ]

Here, column1, column2... are the fields of the existing table and the same would be used to create fields of the new table.

**Example**

Following is an example which would create a table SALARY using the CUSTOMERS table and having the fields – customer ID and customer SALARY:

*SQL*> CREATE TABLE SALARY

AS SELECT ID, SALARY

FROM CUSTOMERS;

**This would create a new table SALARY which will have the following records.**

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

| ID | SALARY  |

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

| 1 | 2000.00  |

| 2 | 1500.00  |

| 3 | 2000.00  |

| 4 | 6500.00  |

| 5 | 8500.00  |

| 6 | 4500.00  |

| 7 | 10000.00  |

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

# SQL ─ DROP or DELETE Table

The SQL DROP TABLE statement is used to remove a table definition and all the data, indexes, triggers, constraints and permission specifications for that table.

NOTE: You should be very careful while using this command because once a table is deleted then all the information available in that table will also be lost forever.

**Syntax**

The basic syntax of this DROP TABLE statement is as follows:

DROP TABLE table\_name;

**Example**

Let us first verify the CUSTOMERS table and then we will delete it from the database as shown below.

*SQL*> DESC CUSTOMERS;

This means that the CUSTOMERS table is available in the database, so let us now drop it as shown below.

*SQL*> DROP TABLE CUSTOMERS;

Query OK, 0 rows affected (0.01 sec)

Now, if you would try the DESC command, then you will get the following error:

*SQL*> DESC CUSTOMERS;

ERROR 1146 (42S02): Table 'TEST.CUSTOMERS' doesn't exist

Here, TEST is the database name which we are using for our examples.

# SQL ─ INSERT Query

The SQL **INSERT INTO** Statement is used to add new rows of data to a table in the database.

**Syntax**

There are two basic syntaxes of the INSERT INTO statement which are shown below.

INSERT INTO TABLE\_NAME (column1, column2, column3,...columnN)]

VALUES (value1, value2, value3,...valueN);

Here, column1, column2, column3,...columnN are the names of the columns in the table into which you want to insert the data.

You may not need to specify the column(s) name in the SQL query if you are adding values for all the columns of the table. But make sure the order of the values is in the same order as the columns in the table.

The **SQL INSERT INTO** syntax will be as follows:

INSERT INTO TABLE\_NAME VALUES (value1,value2,value3,...valueN);

**Example**

The following statements would create six records in the CUSTOMERS table.

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );

You can create a record in the CUSTOMERS table by using the second syntax as shown below.

INSERT INTO CUSTOMERS

VALUES (7, 'Muffy', 24, 'Indore', 10000.00 );

All the above statements would produce the following records in the CUSTOMERS table as shown below.

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

| 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 |

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

**Populate one table using another table**

You can populate the data into a table through the select statement over another table; provided the other table has a set of fields, which are required to populate the first table.

Here is the syntax:

INSERT INTO first\_table\_name [(column1, column2, ... columnN)]

SELECT column1, column2, ...columnN

FROM second\_table\_name

[WHERE condition];

# SQL ─ SELECT Query

The SQL **SELECT** statement is used to fetch the data from a database table which returns this data in the form of a result table. These result tables are called result-sets.

**Syntax**

The basic syntax of the SELECT statement is as follows.:

SELECT column1, column2, columnN FROM table\_name;

Here, column1, column2... are the fields of a table whose values you want to fetch. If you want to fetch all the fields available in the field, then you can use the following syntax.

SELECT \* FROM table\_name;

**Example**

Consider the CUSTOMERS table having the following records:

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

| ID | NAME  | AGE | ADDRESS  | SALARY   |

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

| 1 | Ramesh   | 32  | Ahmedabad | 2000.00  |

| 2 | Khilan   | 25  | Delhi  | 1500.00  |

| 3 | kaushik  | 23  | Kota   | 2000.00  |

| 4 | Chaitali  | 25  | Mumbai   | 6500.00  |

| 5 | Hardik   | 27  | Bhopal   | 8500.00  |

| 6 | Komal  | 22  | MP   | 4500.00  |

| 7 | Muffy  | 24  | Indore   | 10000.00 |

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

The following code is an example, which would fetch the ID, Name and Salary fields of the customers available in CUSTOMERS table.

*SQL*> SELECT ID, NAME, SALARY FROM CUSTOMERS;

This would produce the following result:

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

| ID | NAME  | SALARY   |

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

| 1 | Ramesh   | 2000.00  |

| 2 | Khilan   | 1500.00  |

| 3 | kaushik  | 2000.00  |

| 4 | Chaitali  | 6500.00  |

| 5 | Hardik   | 8500.00  |

| 6 | Komal  | 4500.00  |

| 7 | Muffy  | 10000.00 |

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

If you want to fetch all the fields of the CUSTOMERS table, then you should use the following query.

*SQL*> SELECT \* FROM CUSTOMERS;

This would produce the result as shown below.

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

| 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 |

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

# SQL ─ WHERE Clause

The SQL **WHERE** clause is used to specify a condition while fetching the data from a single table or by joining with multiple tables. If the given condition is satisfied, then only it returns a specific value from the table. You should use the WHERE clause to filter the records and fetching only the necessary records.

The WHERE clause is not only used in the SELECT statement, but it is also used in the UPDATE, DELETE statement, etc., which we would examine in the subsequent chapters.

**Syntax**

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

SELECT column1, column2, columnN

FROM table\_name

WHERE [condition]

You can specify a condition using the comparison or logical operators like **>**, **<**, **=**, **LIKE**, **NOT**, etc. The following examples would make this concept clear.

**Example**

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)

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

*SQL*> SELECT ID, NAME, SALARY

FROM CUSTOMERS

WHERE SALARY > 2000;

This would produce the following result:

| 4 | Chaitali  | 6500.00 |

| 5 | Hardik   | 8500.00 |

| 6 | Komal  | 4500.00 |

| 7 | Muffy  | 10000.00|

The following query is an example, which would fetch the ID, Name and Salary fields from the CUSTOMERS table for a customer with the name **Hardik**.

Here, it is important to note that all the strings should be given inside single quotes (''). Whereas, numeric values should be given without any quote as in the above example.

*SQL*> SELECT ID, NAME, SALARY FROM CUSTOMERS

WHERE NAME = 'Hardik';

This would produce the following result:

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

| ID | NAME  | SALARY  |

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

| 5 | Hardik   | 8500.00 |

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

# SQL ─ AND & OR Conjunctive Operators

The SQL **AND** & **OR** operators are used to combine multiple conditions to narrow data in an SQL statement. These two operators are called as the conjunctive operators.

These operators provide a means to make multiple comparisons with different operators in the same SQL statement.

## The AND Operator

The **AND** operator allows the existence of multiple conditions in an SQL statement's WHERE clause.

**Syntax**

The basic syntax of the AND operator with a WHERE clause is as follows:

SELECT column1, column2, columnN

FROM table\_name

WHERE [condition1] AND [condition2]...AND [conditionN];

You can combine N number of conditions using the AND operator. For an action to be taken by the SQL statement, whether it be a transaction or a query, all conditions separated by the AND must be TRUE.

**Example**

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)

Following is an example, which would fetch the ID, Name and Salary fields from the CUSTOMERS table, where the salary is greater than 2000 and the age is less than 25 years.

*SQL*> SELECT ID, NAME, SALARY

FROM CUSTOMERS

WHERE SALARY > 2000 AND age < 25;

This would produce the following result:

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

| ID | NAME | SALARY   |

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

| 6 | Komal | 4500.00 |

| 7 | Muffy | 10000.00 |

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

## The OR Operator

The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.

**Syntax**

The basic syntax of the OR operator with a WHERE clause is as follows:

SELECT column1, column2, columnN

FROM table\_name

WHERE [condition1] OR [condition2]...OR [conditionN]

You can combine N number of conditions using the OR operator. For an action to be taken by the SQL statement, whether it be a transaction or query, the only any ONE of the conditions separated by the OR must be TRUE.

**Example**

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)

The following code block has a query, which would fetch the ID, Name and Salary fields from the CUSTOMERS table, where the salary is greater than 2000 OR the age is less than 25 years.

*SQL*> SELECT ID, NAME, SALARY FROM CUSTOMERS

WHERE SALARY > 2000 OR age < 25;

This would produce the following result:

| 3 | kaushik  | 2000.00 |

| 4 | Chaitali  | 6500.00 |

| 5 | Hardik   | 8500.00 |

| 6 | Komal  | 4500.00 |

| 7 | Muffy  | 10000.00|

# SQL ─ Using Joins

The SQL **Joins** clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

Consider the following two tables:

**Table 1:** CUSTOMERS Table

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

| 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|

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

**Table 2:** ORDERS Table

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

|OID  | DATE   | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

Now, let us join these two tables in our SELECT statement as shown below.

*SQL*> SELECT ID, NAME, AGE, AMOUNT

FROM CUSTOMERS, ORDERS

WHERE CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result.

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

| ID | NAME  | AGE | AMOUNT  |

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

| 3 | kaushik  | 23  | 3000.00 |

| 3 | kaushik  | 23  | 1500.00 |

| 2 | Khilan   | 25  | 1560.00 |

| 4 | Chaitali  | 25  | 2060.00 |

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

Here, it is noticeable that the join is performed in the WHERE clause. Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; they can all be used to join tables. However, the most common operator is the equal to symbol.

There are different types of joins available in SQL:

* INNER JOIN: returns rows when there is a match in both tables.
* LEFT JOIN: returns all rows from the left table, even if there are no matches in the right table.
* RIGHT JOIN: returns all rows from the right table, even if there are no matches in the left table.
* FULL JOIN: returns rows when there is a match in one of the tables.
* SELF JOIN: is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
* CARTESIAN JOIN: returns the Cartesian product of the sets of records from the two or more joined tables.

Let us now discuss each of these joins in detail.

## SQL - INNER JOIN

The most important and frequently used of the joins is the **INNER JOIN**. They are also referred to as an **EQUIJOIN**.

The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

**Syntax**

The basic syntax of the **INNER JOIN** is as follows.

SELECT table1.column1, table2.column2... FROM table1

INNER JOIN table2

ON table1.common\_field = table2.common\_field;

**Example**

Consider the following two tables:

**Table 1:** CUSTOMERS Table

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

| 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|

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

**Table 2:** ORDERS Table

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

|OID  | DATE   | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

Now, let us join these two tables using the INNER JOIN as follows:

*SQL*> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

INNER JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result.

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

| ID | NAME | AMOUNT | DATE |

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

| 3 | kaushik | 1500 | 2009-10-08 |

| 2 | Khilan | 1560 | 2009-11-20 |

| 3 | kaushik | 3000 | 2009-10-08 |

| 4 | Chaitali | 2060 | 2008-05-20 |

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

## SQL ─ LEFT JOIN

The SQL **LEFT JOIN** returns all rows from the left table, even if there are no matches in the right table. This means that if the ON clause matches 0 (zero) records in the right table; the join will still return a row in the result, but with NULL in each column from the right table.

This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.

**Syntax**

The basic syntax of a **LEFT JOIN** is as follows.

SELECT table1.column1, table2.column2...

FROM table1

LEFT JOIN table2

ON table1.common\_field = table2.common\_field;

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

**Example**

Consider the following two tables:

**Table 1:** CUSTOMERS Table

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

| 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|

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

**Table 2:** ORDERS Table

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

|OID  | DATE   | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

Now, let us join these two tables using the LEFT JOIN as follows.

*SQL*> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

LEFT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result:

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

| ID | NAME | AMOUNT | DATE |

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

| 3 | kaushik | 1500 | 2009-10-08 |

| 2 | Khilan | 1560 | 2009-11-20 |

| 3 | kaushik | 3000 | 2009-10-08 |

| 4 | Chaitali | 2060 | 2008-05-20 |

| 1 | Ramesh | NULL | NULL |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

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

## SQL - RIGHT JOIN

The SQL **RIGHT JOIN** returns all rows from the right table, even if there are no matches in the left table. This means that if the ON clause matches 0 (zero) records in the left table; the join will still return a row in the result, but with NULL in each column from the left table.

This means that a right join returns all the values from the right table, plus matched values from the left table or NULL in case of no matching join predicate.

**Syntax**

The basic syntax of a **RIGHT JOIN** is as follow.

SELECT table1.column1, table2.column2...

FROM table1

RIGHT JOIN table2

ON table1.common\_field = table2.common\_field;

**Example**

Consider the following two tables:

**Table 1:** CUSTOMERS Table

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

| 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|

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

**Table 2:** ORDERS Table

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

|OID  | DATE   | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

Now, let us join these two tables using the RIGHT JOIN as follows.

*SQL*> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

RIGHT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result:

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

| ID | NAME | AMOUNT | DATE |

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

| 3 | kaushik | 1500 | 2009-10-08 |

| 2 | Khilan | 1560 | 2009-11-20 |

| 3 | kaushik | 3000 | 2009-10-08 |

| 4 | Chaitali | 2060 | 2008-05-20 |

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

## SQL ─ FULL JOIN

The SQL **FULL JOIN** combines the results of both left and right outer joins.

The joined table will contain all records from both the tables and fill in NULLs for missing matches on either side.

**Syntax**

The basic syntax of a **FULL JOIN** is as follows:

SELECT table1.column1, table2.column2...

FROM table1

FULL JOIN table2

ON table1.common\_field = table2.common\_field;

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

**Example**

Consider the following two tables:

**Table 1:** CUSTOMERS Table

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

| 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|

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

**Table 2:** ORDERS Table

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

|OID  | DATE   | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

Now, let us join these two tables using FULL JOIN as follows.

*SQL*> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

FULL JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result.

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

| ID   | NAME  | AMOUNT | DATE  |

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

| 1  | Ramesh  | NULL   |  NULL     |

| 2  | Khilan  | 1560   |  2009-11-20  00:00:00  |

| 3  | kaushik  | 3000   |  2009-10-08  00:00:00  |

| 3  | kaushik  | 1500   |  2009-10-08  00:00:00  |

| 4  | Chaitali | 2060   |  2008-05-20  00:00:00  |

| 5  | Hardik  | NULL   |  NULL     |

| 6  | Komal   | NULL   |  NULL     |

| 7  | Muffy   | NULL   |  NULL     |

| 3  | kaushik  | 3000   |  2009-10-08  00:00:00  |

| 3  | kaushik  | 1500   |  2009-10-08  00:00:00  |

| 2  | Khilan  | 1560   |  2009-11-20  00:00:00  |

| 4  | Chaitali | 2060   |  2008-05-20  00:00:00  |

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

**If your Database does not support FULL JOIN (MySQL does not support FULL JOIN), then you can use UNION ALL clause to combine these two JOINS as shown below.**

*SQL*> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

LEFT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID

UNION ALL

SELECT  ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

RIGHT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID

## SQL ─ SELF JOIN

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

**Syntax**

The basic syntax of **SELF JOIN** is as follows:

SELECT a.column\_name, b.column\_name...

FROM table1 a, table1 b

WHERE a.common\_field = b.common\_field;

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

**Example**

Consider the following table.

**CUSTOMERS Table** is as follows.

Consider the following two tables:

**Table 1:** CUSTOMERS Table

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

| 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, let us join this table using SELF JOIN as follows:

*SQL*> SELECT a.ID, b.NAME, a.SALARY

FROM CUSTOMERS a, CUSTOMERS b

WHERE a.SALARY < b.SALARY;

This would produce the following result:

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

| ID | NAME | SALARY |

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

| 2 | Ramesh | 1500.00 |

| 2 | kaushik | 1500.00 |

| 1 | Chaitali | 2000.00 |

| 2 | Chaitali | 1500.00 |

| 3 | Chaitali | 2000.00 |

| 6 | Chaitali | 4500.00 |

| 1 | Hardik | 2000.00 |

| 2 | Hardik | 1500.00 |

| 3 | Hardik | 2000.00 |

| 4 | Hardik | 6500.00 |

| 6 | Hardik | 4500.00 |

| 1 | Komal | 2000.00 |

| 2 | Komal | 1500.00 |

| 3 | Komal | 2000.00 |

| 1 | Muffy | 2000.00 |

| 2 | Muffy | 1500.00 |

| 3 | Muffy | 2000.00 |

| 4 | Muffy | 6500.00 |

| 5 | Muffy | 8500.00 |

| 6 | Muffy | 4500.00 |

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

## SQL ─ CARTESIAN or CROSS JOIN

The CARTESIAN JOIN or CROSS JOIN returns the Cartesian product of the sets of records from two or more joined tables. Thus, it equates to an inner join where the join-condition always evaluates to either True or where the join-condition is absent from the statement.

**Syntax**

The basic syntax of the **CARTESIAN JOIN** or the **CROSS JOIN** is as follows:

SELECT table1.column1, table2.column2...

FROM  table1, table2 [, table3 ]

**Example**

Consider the following two tables:

**Table 1:** CUSTOMERS Table

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

| 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|

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

**Table 2:** ORDERS Table

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

|OID  | DATE   | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

Now, let us join these two tables using INNER JOIN as follows:

*SQL*> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS, ORDERS;

This would produce the following result:

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

| ID | NAME | AMOUNT | DATE |

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

| 1 | Ramesh | 1500 | 2009-10-08 |

| 1 | Ramesh | 1560 | 2009-11-20 |

| 1 | Ramesh | 3000 | 2009-10-08 |

| 1 | Ramesh | 2060 | 2008-05-20 |

| 2 | Khilan | 1500 | 2009-10-08 |

| 2 | Khilan | 1560 | 2009-11-20 |

| 2 | Khilan | 3000 | 2009-10-08 |

| 2 | Khilan | 2060 | 2008-05-20 |

| 3 | kaushik | 1500 | 2009-10-08 |

| 3 | kaushik | 1560 | 2009-11-20 |

| 3 | kaushik | 3000 | 2009-10-08 |

| 3 | kaushik | 2060 | 2008-05-20 |

| 4 | Chaitali | 1500 | 2009-10-08 |

| 4 | Chaitali | 1560 | 2009-11-20 |

| 4 | Chaitali | 3000 | 2009-10-08 |

| 4 | Chaitali | 2060 | 2008-05-20 |

| 5 | Hardik | 1500 | 2009-10-08 |

| 5 | Hardik | 1560 | 2009-11-20 |

| 5 | Hardik | 3000 | 2009-10-08 |

| 5 | Hardik | 2060 | 2008-05-20 |

| 6 | Komal | 1500 | 2009-10-08 |

| 6 | Komal | 1560 | 2009-11-20 |

| 6 | Komal | 3000 | 2009-10-08 |

| 6 | Komal | 2060 | 2008-05-20 |

| 7 | Muffy | 1500 | 2009-10-08 |

| 7 | Muffy | 1560 | 2009-11-20 |

| 7 | Muffy | 3000 | 2009-10-08 |

| 7 | Muffy | 2060 | 2008-05-20 |

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

# SQL ─ Having Clause

The **HAVING Clause** enables you to specify conditions that filter which group results appear in the 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 code block shows 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 code block has 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 an example, which would display a record for a similar age count that would be more than or equal to 2.

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

FROM CUSTOMERS

GROUP BY age

HAVING COUNT(age) >= 2;

This would produce the following result:

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

| ID | NAME  | AGE | ADDRESS | SALARY  |

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

| 2 | Khilan | 25 | Delhi  | 1500.00 |

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

# 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.

A transaction is the propagation of one or more changes to the database. For example, if you are creating a record or updating a record or deleting a record from the table, then you are performing a transaction on that table. It is important to control these transactions to ensure the data integrity and to handle database errors.

Practically, you will club many SQL queries into a group and you will execute all of them together as a part of a transaction.

## 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 all the 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**

The following commands are used to control transactions.

* COMMIT: to save the changes.
* ROLLBACK: to roll back the changes.
* SAVEPOINT: creates points within the groups of transactions in which to ROLLBACK.
* SET TRANSACTION**:** Places a name on a transaction.

## Transactional Control Commands

Transactional control commands are only used with the **DML Commands** such as – INSERT, UPDATE and DELETE only. They cannot be used while creating tables or dropping them because these operations are automatically committed 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 the transactions to the database since the last COMMIT or ROLLBACK command.

The syntax for the 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 an example which would delete those records from the table which have age = 25 and then COMMIT the changes in the database.

*SQL*> DELETE FROM CUSTOMERS

WHERE AGE = 25;

*SQL*> COMMIT;

Thus, two rows from the table would be deleted and the 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. This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

The syntax for a 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 an example, which would delete those records from the table which have the age = 25 and then ROLLBACK the changes in the database.

*SQL*> DELETE FROM CUSTOMERS

WHERE AGE = 25;

*SQL*> ROLLBACK;

Thus, the delete operation would not impact the table and the 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 a SAVEPOINT command is as shown below.

SAVEPOINT SAVEPOINT\_NAME;

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

The syntax for rolling back to a SAVEPOINT is as shown below.

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|

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

The following code block contains 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, let us assume that 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.

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

| 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|

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

### The RELEASE SAVEPOINT Command

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

The syntax for a RELEASE SAVEPOINT command 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 last 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 a SET TRANSACTION command is as follows.

SET TRANSACTION [ READ WRITE | READ ONLY ];