SQL Tutorial

Simply Easy Learning by Bindu

ABOUT THE TUTORIAL

SQL Tutorial

SQL is a database computer language designed for the retrieval and management of data in relational database. SQL stands for Structured Query Language.

This tutorial will give you quick start with SQL.

Audience

This reference has been prepared for the beginners to help them understand the basic to advanced concepts related to SQL languages.

Prerequisites

Before you start doing practice with various types of examples given in this reference, I'm making an assumption that you are already aware about what is database, especially RDBMS and what is a computer programming language.

Contents

CHAPTER

1

SQL Overview

SQL tutorial gives unique learning on **Structured Query Language** and it helps to make practice on SQL commands which provides immediate results. SQL is a language of database, it includes database creation, deletion, fetching rows and modifying rows etc.

SQL is an ANSI (American National Standards Institute) standard, 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 relational database.

SQL is the standard language for Relation Database System. All relational database management systems like MySQL, MS Access, Oracle, Sybase, Informix, postgres and SQL Server use SQL as 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?

- Allows users to access data in relational database management systems.
- Allows users to describe the data.

- Allows users to define the data in 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

History:

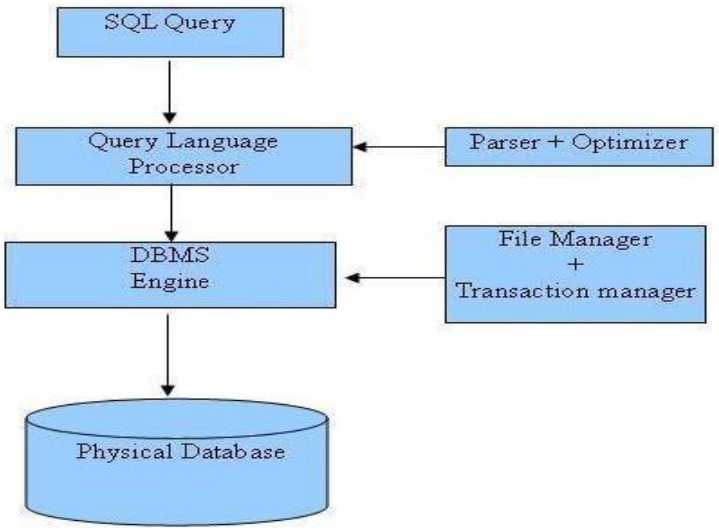
- **1970 --** Dr. E. F. "Ted" of IBM is known as the father of relational databases. He described a relational model for databases.
- **1974 --** Structured Query Language appeared.
- **1978 --** IBM worked to develop Codd's ideas and released a product named System/R.
- **1986 --** IBM developed the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

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:

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.

DML - Data Manipulation Language:

Command	Description
INSERT	Creates a record
UPDATE	Modifies records
DELETE	Deletes records

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

CHAPTER

2

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 table?

The data in RDBMS is stored in database objects called **tables**. The table is a collection of related data entries and it consists of columns and rows.

Remember, a table is the most common and simplest form of data storage in a relational database. Following is the example of a 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 |

+----+----------+-----+------------+----------+
```

What is 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 record or row?

A record, also called 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 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 consist of the following:

```
+------------+

| ADDRESS    |

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

| Ahmedabad |

| Delhi     |

| Kota      |

| Mumbai    |

| Indore    |

+-----+------+
```

What is 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 one that has been left blank during record creation.

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:

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

NOT NULL Constraint:

By default, a column can hold NULL values. If you do not want a column to have a NULL value, then you need to define such constraint on this column specifying that NULL is now not allowed for that column.

A NULL is not the same as no data, rather, it represents unknown data.

Example:

For example, the following SQL creates a new table called CUSTOMERS and adds five columns, three of which, ID and NAME and AGE, specify not to accept NULLs:

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

If CUSTOMERS table has already been created, then to add a NOT NULL constraint to SALARY column in Oracle and MySQL, you would write a statement similar to the following:

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

DEFAULT Constraint:

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

Example:

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

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

If CUSTOMERS table has already been created, then to add a DFAULT constraint to SALARY column, you would write a statement similar to the following:

```
ALTER TABLE CUSTOMERS

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

Drop Default Constraint:

To drop a DEFAULT constraint, use the following SQL:

```
ALTER TABLE CUSTOMERS

   ALTER COLUMN SALARY DROP DEFAULT;
```

UNIQUE Constraint:

The UNIQUE Constraint prevents two records from having identical values in a particular column. In the CUSTOMERS table, for example, you might want to prevent two or more people from having identical age.

Example:

For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, AGE column is set to UNIQUE, so that you can not have two records with same age:

```
CREATE TABLE CUSTOMERS(

       ID   INT              NOT NULL,
```

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

If CUSTOMERS table has already been created, then to add a UNIQUE constraint to AGE column, you would write a statement similar to the following:

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

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

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

DROP a UNIQUE Constraint:

To drop a UNIQUE constraint, use the following SQL:

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

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

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

PRIMARY Key:

A primary key is a field in a table which uniquely identifies each row/record in a database table. Primary keys must contain unique values. A primary key column cannot have NULL values.

A table can have only one primary key, which may consist of single or multiple fields. When multiple fields are used as a primary key, they are called a **composite key**.

If a table has a primary key defined on any field(s), then you can not have two records having the same value of that field(s).

Note: You would use these concepts while creating database tables.

Create Primary Key:

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

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

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

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

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

For defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

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

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

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

Delete Primary Key:

You can clear the primary key constraints from the table, Use Syntax:

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

FOREIGN Key:

A foreign key is a key used to link two tables together. This is sometimes called a referencing key.

Foreign Key is a column or a combination of columns whose values match a Primary Key in a different table.

The relationship between 2 tables matches the Primary Key in one of the tables with a Foreign Key in the second table.

If a table has a primary key defined on any field(s), then you can not have two records having the same value of that field(s).

Example:

Consider the structure of the two tables as follows:

CUSTOMERS table:

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

ORDERS table:

```
CREATE TABLE ORDERS (
       ID          INT        NOT NULL,
       DATE        DATETIME,
       CUSTOMER_ID INT references CUSTOMERS(ID),
       AMOUNT     double,
       PRIMARY KEY (ID)
);
```

If ORDERS table has already been created, and the foreign key has not yet been set, use the syntax for specifying a foreign key by altering a table.

```
ALTER TABLE ORDERS
   ADD FOREIGN KEY (Customer_ID) REFERENCES CUSTOMERS (ID);
```

DROP a FOREIGN KEY Constraint:

To drop a FOREIGN KEY constraint, use the following SQL:

```
ALTER TABLE ORDERS

   DROP FOREIGN KEY;
```

CHECK Constraint:

The CHECK Constraint enables a condition to check the value being entered into a record. If the condition evaluates to false, the record violates the constraint and isn't entered into the table.

Example:

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

```
CREATE TABLE CUSTOMERS(

       ID   INT              NOT NULL,

       NAME VARCHAR (20)     NOT NULL,

       AGE  INT              NOT NULL CHECK (AGE >= 18),

       ADDRESS  CHAR (25) ,

       SALARY   DECIMAL (18, 2),

       PRIMARY KEY (ID)

);
```

If CUSTOMERS table has already been created, then to add a CHECK constraint to AGE column, you would write a statement similar to the following:

```
ALTER TABLE CUSTOMERS

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

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

```
ALTER TABLE CUSTOMERS

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

DROP a CHECK Constraint:

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

```
ALTER TABLE CUSTOMERS

   DROP CONSTRAINT myCheckConstraint;
```

INDEX:

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

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

Example:

For example, the following SQL creates a new table called CUSTOMERS and adds five columns:

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

Now, you can create index on single or multiple columns using the following syntax:

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

To create an INDEX on AGE column, to optimize the search on customers for a particular age, following is the SQL syntax:

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

DROP an INDEX Constraint:

To drop an INDEX constraint, use the following SQL:

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

Data Integrity:

The following categories of the 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 the normalization process:

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

Both of these are worthy goals as they reduce the amount of space a database consumes and ensure 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 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 third normal form.

It's your choice to take it further and go to fourth normal form, fifth normal form, and so on, but generally speaking, third normal form is enough.

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

First Normal Form

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

- Define the data items required, because they become the columns in a table. Place 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 finally putting 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)     NOT NULL,
       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 follows:

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 1NF, we need to ensure that there are no repeating groups of data. So let us break above table into two parts and join them using a key as follows:

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.

Second Normal Form

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 date of purchase:

```
CREATE TABLE CUSTOMERS(

       CUST_ID    INT              NOT NULL,

       CUST_NAME VARCHAR (20)      NOT NULL,

       ORDER_ID   INT              NOT NULL,        ORDER_DETAIL VARCHAR (20)  NOT
NULL,

       SALE_DATE  DATETIME,

       PRIMARY KEY (CUST_ID, ORDER_ID)

);
```

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

However, the table is not in 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. Order detail and purchase date are also dependent on ORDER_ID, but they are not dependent on CUST_ID, because there's no link between a CUST_ID and an ORDER_DETAIL or their SALE_DATE.

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

First, create a table to store the customer details as follows:

```
CREATE TABLE CUSTOMERS(
       CUST_ID    INT              NOT NULL,        CUST_NAME VARCHAR (20)      NOT
NULL,
       PRIMARY KEY (CUST_ID)
);
```

Next, create a table to store 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 CUST_ID and ORDER_ID to keep 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)
);
```

Third Normal Form

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

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

The dependency of nonprimary fields is between the data. For example, in the below table, street name, city, and state are unbreakably bound to the 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 zip code and address is called a transitive dependency. To comply with third normal form, all you need to do is move the Street, City, and State fields into their own table, which you can call the Zip Code table:

```
CREATE TABLE ADDRESS(

       ZIP           VARCHAR(12),

       STREET        VARCHAR(200),

       CITY          VARCHAR(100),

       STATE         VARCHAR(100),

       PRIMARY KEY (ZIP)

);
```

Next, alter the CUSTOMERS table as follows:

```
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 twofold. 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's a big risk of updating only some of the data, especially if it's spread out in a number of different places in the database. For example, if address and zip code data were stored in three or four different tables, then any changes in zip codes would need to ripple out to every record in those three or four tables.

CHAPTER

3

SQL RDBMS Databases

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

MySQL

MySQL is an open source SQL database, which is developed by Swedish company MySQL AB. MySQL is pronounced "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 23 May 1995.
- Windows version was released on 8 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 26 February 2008.
- Version 5.1: production release 27 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 and multi-user database management system. Oracle is a relational database management system developed by 'Oracle Corporation'.

Oracle works to efficiently manage its resource, 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 has 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 an inexpensive but also 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 of 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.
- The import and export of 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 client-server relational database management systems (RDBMS), Microsoft Access does not implement database triggers, stored procedures, or transaction logging.

CHAPTER

4

SQL Syntax

SQL is followed by 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 (;).

Important point to be noted is that SQL is **case insensitive,** which means SELECT and select have same meaning in SQL statements, but 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.

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 database_name;
```

SQL COMMIT Statement:

```
COMMIT;
```

SQL ROLLBACK Statement:

```
ROLLBACK;
```

CHAPTER

5

SQL Data Types

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

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

SQL Server offers six categories of data types for your use:

Exact Numeric Data Types:

DATA TYPE	FROM	TO

Bigint	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
Int	-2,147,483,648	2,147,483,647
Smallint	-32,768	32,767
Tinyint	0	255
Bit	0	1
Decimal	-10^38 +1	10^38 -1
Numeric	-10^38 +1	10^38 -1
Money	-922,337,203,685,477.5808	+922,337,203,685,477.5807
Smallmoney	-214,748.3648	+214,748.3647

Approximate Numeric Data Types:

DATA TYPE	FROM	TO
Float	-1.79E + 308	1.79E + 308
Real	-3.40E + 38	3.40E + 38

Date and Time Data Types:

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

Note: Here, datetime has 3.33 milliseconds accuracy where as smalldatetime has 1 minute accuracy.

Character Strings Data Types:

DATA TYPE	FROM	TO
Char	Char	Maximum length of 8,000 characters.(Fixed length non-Unicode characters)
Varchar	Varchar	Maximum of 8,000 characters.(Variable-length non-Unicode data).
varchar(max)	varchar(max)	Maximum length of 231characters, Variable-length non-Unicode data (SQL Server 2005 only).
Text	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

CHAPTER

6

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.

Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

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

SQL Arithmetic Operators:

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

Operator	Description	Example
+	Addition - Adds values on either side of the operator	a + b will give 30

-	Subtraction - Subtracts right hand operand from left hand operand	a - b will give -10
*	Multiplication - Multiplies values on either side of the operator	a * b will give 200
/	Division - Divides left hand operand by right hand operand	b / a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	b % a will give 0

Here are simple examples showing usage of SQL Arithmetic Operators:

```
SQL> select 10+ 20;
+-----------+
| 10+ 20 |
+-----------+
|      30 |
+-----------+
1 row in set (0.00 sec)

SQL> select 10 * 20;
+------------+
| 10 * 20 |
+------------+
|      200 |
+------------+
1 row in set (0.00 sec)

SQL> select 10 / 5;
+-----------+
| 10 / 5 |
+-----------+
| 2.0000 |
+-----------+
1 row in set (0.03 sec)

SQL> select 12 %  5;
+------------+
| 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.

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 simple examples showing usage of SQL Comparison Operators:

```
SQL> SELECT * FROM CUSTOMERS WHERE SALARY > 5000;
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
|  4 | Chaitali |  25 | Mumbai  |  6500.00 |
```

```
|  5 | Hardik    |  27 | Bhopal   |  8500.00 |
|  7 | Muffy     |  24 | Indore   | 10000.00 |
+-----+-------------+-------+------------+-------------+
3 rows in set (0.00 sec)

SQL>  SELECT * FROM CUSTOMERS WHERE SALARY = 2000;
+-----+------------+-------+--------------+------------+
| ID | NAME     | AGE | ADDRESS    | SALARY   |
+-----+------------+-------+--------------+------------+
|  1 | Ramesh   |  32 | Ahmedabad | 2000.00 |
|  3 | kaushik |  23 | Kota       | 2000.00 |
+-----+------------+-------+--------------+------------+
2 rows in set (0.00 sec)

SQL>  SELECT * FROM CUSTOMERS WHERE SALARY != 2000;
+-----+-------------+-------+------------+-------------+
| 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 |
+-----+-------------+-------+------------+-------------+
5 rows in set (0.00 sec)

SQL> SELECT * FROM CUSTOMERS WHERE SALARY <> 2000;
+-----+-------------+-------+------------+-------------+
| 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 |
+-----+-------------+-------+------------+-------------+
5 rows in set (0.00 sec)
SQL> SELECT * FROM CUSTOMERS WHERE SALARY >= 6500;
+-----+-------------+-------+------------+-------------+
| ID | NAME      | AGE | ADDRESS | SALARY    |
+-----+-------------+-------+------------+-------------+
|  4 | Chaitali |  25 | Mumbai   |  6500.00 |
|  5 | Hardik    |  27 | Bhopal   |  8500.00 |
|  7 | Muffy     |  24 | Indore   | 10000.00 |
+-----+-------------+-------+------------+-------------+
3 rows in set (0.00 sec)
```

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 according to 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 certain criteria.
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).

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 simple examples showing usage of SQL Comparison Operators:

```
SQL> SELECT * FROM CUSTOMERS WHERE AGE >= 25 AND SALARY >= 6500;
+------+--------------+-------+------------+------------+
| ID | NAME      | AGE | ADDRESS | SALARY   |
```

```
+------+-------------+-------+------------+------------+
|  4 | Chaitali |  25 | Mumbai   | 6500.00 |
|  5 | Hardik   |  27 | Bhopal   | 8500.00 |
+------+-------------+-------+------------+------------+
2 rows in set (0.00 sec)

SQL> SELECT * FROM CUSTOMERS WHERE AGE >= 25 OR SALARY >= 6500;
+------+-------------+-------+--------------+-------------+
| 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 |
+------+-------------+-------+--------------+-------------+
5 rows in set (0.00 sec)
SQL>  SELECT * FROM CUSTOMERS WHERE AGE IS NOT NULL;
+------+-------------+-------+--------------+-------------+
| 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)

SQL> SELECT * FROM CUSTOMERS WHERE NAME LIKE 'Ko%';
+------+---------+-------+------------+------------+
| ID | NAME   | AGE | ADDRESS | SALARY   |
+------+---------+-------+------------+------------+
|  6 | Komal |  22 | MP       | 4500.00 |
+------+---------+-------+------------+------------+
1 row in set (0.00 sec)

SQL> SELECT * FROM CUSTOMERS WHERE AGE IN ( 25, 27 );
+------+-------------+-------+------------+------------+
| ID | NAME      | AGE | ADDRESS | SALARY   |
+------+-------------+-------+------------+------------+
|  2 | Khilan    |  25 | Delhi    | 1500.00 |
|  4 | Chaitali |  25 | Mumbai   | 6500.00 |
|  5 | Hardik    |  27 | Bhopal   | 8500.00 |
+------+-------------+-------+------------+------------+
3 rows in set (0.00 sec)

SQL> SELECT * FROM CUSTOMERS WHERE AGE BETWEEN 25 AND 27;
+------+-------------+-------+------------+------------+
| ID | NAME      | AGE | ADDRESS | SALARY   |
+------+-------------+-------+------------+------------+
|  2 | Khilan    |  25 | Delhi    | 1500.00 |
|  4 | Chaitali |  25 | Mumbai   | 6500.00 |
```

```
|  5 | Hardik    |  27 | Bhopal   | 8500.00 |
+-----+-------------+-------+-----------+-----------+
3 rows in set (0.00 sec)

SQL> SELECT AGE FROM CUSTOMERS
WHERE EXISTS (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);
+     +
| AGE |
+-------+
|  32 |
|  25 |
|  23 |
|  25 |
|  27 |
|  22 |
|  24 |
+-------+
7 rows in set (0.02 sec)

SQL> SELECT * FROM CUSTOMERS
WHERE AGE > ALL (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);
+-----+----------+-------+-------------+-----------+
| ID | NAME    | AGE | ADDRESS    | SALARY   |
+-----+----------+-------+-------------+-----------+
|  1 | Ramesh |  32 | Ahmedabad | 2000.00 |
+-----+----------+-------+-------------+-----------+
1 row in set (0.02 sec)

SQL> SELECT * FROM CUSTOMERS
WHERE AGE > ANY (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);
+-----+-------------+-------+-------------+-----------+
| 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 |
+-----+-------------+-------+-------------+-----------+
4 rows in set (0.00 sec)
```

CHAPTER

7

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 MATCHTING 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 |
+---------------------------------+
1 row in set (0.00 sec)
```

CHAPTER

8

SQL CREATE Database

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

Syntax:

Basic syntax of CREATE DATABASE statement is as follows:

```
CREATE DATABASE DatabaseName;
```

Always database name should be unique within the RDBMS.

Example:

If you want to create new database <testDB>, then CREATE DATABASE statement would be as follows:

```
SQL> CREATE DATABASE testDB;
```

Make sure you have 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)
```

CHAPTER

9

DROP or DELETE Database

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

Syntax:

Basic syntax of DROP DATABASE statement is as follows:

```
DROP DATABASE DatabaseName;
```

Always database name should be unique within the RDBMS.

Example:

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

```
SQL> DROP DATABASE testDB;
```

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

Make sure you have admin privilege before dropping any database. Once a database is dropped, you can check it.

```
SQL> in the list of databases as follows:SHOW DATABASES;
+----------------------------+
| Database                   |
+----------------------------+
```

```
| information_schema |
| AMROOD             |
| TUTORIALSPOINT     |
| mysql              |
| orig               |
| test               |
+--------------------+
6 rows in set (0.00 sec)
```

CHAPTER

10

SQL SELECT Database

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 SQL schema.

Syntax:

Basic syntax of USE statement is as follows:

```
USE DatabaseName;
```

Always database name should be unique within the RDBMS.

Example:

You can check available databases as follows:

```
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 AMROOD database, then you can execute the following SQL command and start working with AMROOD database:

```
SQL> USE AMROOD;
```

CHAPTER

11

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:

Basic syntax of 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 an example below.

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

Create Table Using another 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 create a new table using existing table, new table would be populated using 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 existing table and same would be used to create fields of new table.

Example:

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

```
SQL> CREATE TABLE SALARY
AS
   SELECT ID, SALARY
FROM CUSTOMERS;
```

This would create new table SALARY, which would 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 |
+    +          +
```

Example:

Following is an example, which creates a CUSTOMERS table with ID as primary key and NOT NULL are the constraints showing that these fileds can not 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 **DESC** command as follows:

```
SQL> DESC CUSTOMERS;
+------------+--------------------+--------+-------+------------+---------+
| Field      | Type               | Null   | Key   | Default    | Extra   |
+------------+--------------------+--------+-------+------------+---------+
| ID         | int(11)            | NO     | PRI   |            |         |
| NAME       | varchar(20)        | NO     |       |            |         |
| AGE        | int(11)            | NO     |       |            |         |
| ADDRESS    | char(25)           | YES    |       | NULL       |         |
| SALARY     | decimal(18,2)      | YES    |       | NULL       |         |
+------------+--------------------+--------+-------+------------+---------+
5 rows in set (0.00 sec)
```

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

CHAPTER

12

SQL DROP or DELETE Table

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

NOTE: You have to be careful while using this command because once a table is deleted then all the information available in the table would also be lost forever.

Syntax:

Basic syntax of DROP TABLE statement is as follows:

```
DROP TABLE table_name;
```

Example:

Let us first verify CUSTOMERS table and then we would delete it from the database:

```
SQL> DESC CUSTOMERS;
+------------+--------------------+--------+-------+------------+----------+
| Field      | Type               | Null   | Key   | Default    | Extra    |
+------------+--------------------+--------+-------+------------+----------+
| ID         | int(11)            | NO     | PRI   |            |          |
| NAME       | varchar(20)        | NO     |       |            |          |
| AGE        | int(11)            | NO     |       |            |          |
| ADDRESS    | char(25)           | YES    |       | NULL       |          |
| SALARY     | decimal(18,2)      | YES    |       | NULL       |          |
+------------+--------------------+--------+-------+------------+----------+
5 rows in set (0.00 sec)
```

This means CUSTOMERS table is available in the database, so let us drop it as follows:

```
SQL> DROP TABLE CUSTOMERS;
Query OK, 0 rows affected (0.01 sec)
```

Now, if you would try DESC command, then you would get error as follows:

```
SQL> DESC CUSTOMERS;
ERROR 1146 (42S02): Table 'TEST.CUSTOMERS' doesn't exist
```

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

CHAPTER

13

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 INSERT INTO statement as follows:

```
INSERT INTO TABLE_NAME (column1, column2, column3,...columnN)]
VALUES (value1, value2, value3,...valueN);
```

Here, column1, column2,...columnN are the names of the columns in the table into which you want to insert 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 would be as follows:

```
INSERT INTO TABLE_NAME VALUES (value1,value2,value3,...valueN);
```

Example:

Following statements would create six records in 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 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Komal', 22, 'MP', 4500.00 );
```

You can create a record in CUSTOMERS table using second syntax as follows:

```
INSERT INTO CUSTOMERS
VALUES (7, 'Muffy', 24, 'Indore', 10000.00 );
```

All the above statements would produce the following records in 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   |
+-----+-------------+-------+--------------+-------------+
```

Populate one table using another table:

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

```
INSERT INTO first_table_name [(column1, column2, ... columnN)]
   SELECT column1, column2, ...columnN
   FROM second_table_name
   [WHERE condition];
```

CHAPTER

14

SQL SELECT Query

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

Syntax:

The basic syntax of 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 |
+    +          +     +           +          +
```

Following is an example, which would fetch 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 CUSTOMERS table, then use the following query:

```
SQL> SELECT * 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 |
|  4  | Chaitali |  25   | Mumbai    |  6500.00 |
|  5  | Hardik   |  27   | Bhopal    |  8500.00 |
|  6  | Komal    |  22   | MP        |  4500.00 |
|  7  | Muffy    |  24   | Indore    | 10000.00 |
+-----+----------+-------+-----------+----------+
```

CHAPTER

15

SQL WHERE Clause

The SQL **WHERE** clause is used to specify a condition while fetching the data from single table or joining with multiple tables.

If the given condition is satisfied, then only it returns specific value from the table. You would use WHERE clause to filter the records and fetching only necessary records.

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

Syntax:

The basic syntax of SELECT statement with WHERE clause is as follows:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition]
```

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

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 fetch ID, Name and Salary fields from the CUSTOMERS table where salary is greater than 2000:

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

This would produce the following result:

```
+----+-----------+----------+
| ID | NAME      | SALARY   |
+----+-----------+----------+
|  4 | Chaitali  |  6500.00 |
|  5 | Hardik    |  8500.00 |
|  6 | Komal     |  4500.00 |
|  7 | Muffy     | 10000.00 |
+----+-----------+----------+
```

Following is an example, which would fetch ID, Name and Salary fields from the CUSTOMERS table for a customer with name **Hardik**. Here, it is important to note that all the strings should be given inside single quotes (') where as numeric values should be given without any quote as in 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 |
+----+-----------+----------+
```

CHAPTER

16

SQL AND and OR Operators

The SQL **AND** and **OR** operators are used to combine multiple conditions to narrow data in an SQL statement. These two operators are called 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 AND operator with 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 AND operator. For an action to be taken by the SQL statement, whether it be a transaction or query, all conditions separated by the AND must be TRUE.

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 fetch ID, Name and Salary fields from the CUSTOMERS table where salary is greater than 2000 AND age is less tan 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 OR operator with 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 OR operator. For an action to be taken by the SQL statement, whether it be a transaction or query, only any ONE of the conditions separated by the OR must be TRUE.

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 fetch ID, Name and Salary fields from the CUSTOMERS table where salary is greater than 2000 OR age is less tan 25 years:

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000 OR age < 25;
```

This would produce the following result:

```
+------+--------------+--------------+
| ID | NAME      | SALARY    |
+------+--------------+--------------+
|  3 | kaushik   |  2000.00 |
|  4 | Chaitali |  6500.00 |
```

```
|   5 | Hardik    |  8500.00 |
|   6 | Komal     |  4500.00 |
|   7 | Muffy     | 10000.00 |
+-----+-----------+----------+
```

CHAPTER

17

SQL UPDATE Query

The SQL **UPDATE** Query is used to modify the existing records in a table.

You can use WHERE clause with UPDATE query to update selected rows, otherwise all the rows would be affected.

Syntax:

The basic syntax of UPDATE query with WHERE clause is as follows:

```
UPDATE table_name
SET column1 = value1, column2 = value2...., columnN = valueN
WHERE [condition];
```

You can combine N number of conditions using AND or OR operators.

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 update ADDRESS for a customer whose ID is 6:

```
SQL> UPDATE
CUSTOMERS
SET ADDRESS = 'Pune'
WHERE ID = 6;
```

Now, CUSTOMERS table would have 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 | Pune      |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

If you want to modify all ADDRESS and SALARY column values in CUSTOMERS table, you do not need to use WHERE clause and UPDATE query would be as follows:

```
SQL> UPDATE CUSTOMERS
SET ADDRESS = 'Pune', SALARY = 1000.00;
```

Now, CUSTOMERS table would have the following records:

```
+----+----------+-----+---------+---------+
| ID | NAME     | AGE | ADDRESS | SALARY  |
+----+----------+-----+---------+---------+
|  1 | Ramesh   |  32 | Pune    | 1000.00 |
|  2 | Khilan   |  25 | Pune    | 1000.00 |
|  3 | kaushik  |  23 | Pune    | 1000.00 |
|  4 | Chaitali |  25 | Pune    | 1000.00 |
|  5 | Hardik   |  27 | Pune    | 1000.00 |
|  6 | Komal    |  22 | Pune    | 1000.00 |
|  7 | Muffy    |  24 | Pune    | 1000.00 |
+----+----------+-----+---------+---------+
```

CHAPTER

18

SQL DELETE Query

The SQL **DELETE** Query is used to delete the existing records from a table.

You can use WHERE clause with DELETE query to delete selected rows, otherwise all the records would be deleted.

Syntax:

The basic syntax of DELETE query with WHERE clause is as follows:

```
DELETE FROM table_name
WHERE [condition];
```

You can combine N number of conditions using AND or OR operators.

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 a customer, whose ID is 6:

```
SQL> DELETE FROM CUSTOMERS
WHERE ID = 6;
```

Now, CUSTOMERS table would have 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 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+    +          +     +           +          +
```

If you want to DELETE all the records from CUSTOMERS table, you do not need to use WHERE clause and DELETE query would be as follows:

```
SQL> DELETE FROM CUSTOMERS;
```

Now, CUSTOMERS table would not have any record.

CHAPTER

19

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    |
+-----+-------------+-------+-------------+-------------+
```

CHAPTER

20

SQL TOP 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 |
+------+------------+-------+--------------+------------+
```

CHAPTER

21

SQL ORDER BY Clause

The SQL **ORDER BY** clause is used to sort the data in ascending or descending order, based on one or more columns. Some database sorts query results in ascending order by default.

Syntax:

The basic syntax of ORDER BY clause is as follows:

```
SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

You can use more than one column in the ORDER BY clause. Make sure whatever column you are using to sort, that column should be in column-list.

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 sort the result in ascending order by NAME and SALARY:

```
SQL> SELECT * FROM CUSTOMERS
     ORDER BY NAME, SALARY;
```

This would produce the following result:

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

Following is an example, which would sort the result in descending order by NAME:

```
SQL> SELECT * FROM CUSTOMERS       ORDER BY NAME DESC;
```

This would produce the following result:

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

CHAPTER

22

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 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 |
+          +             +
```

CHAPTER

23

SQL Distinct Keyword

The SQL **DISTINCT** keyword is used in conjunction with SELECT statement to eliminate all the duplicate records and fetching only unique records.

There may be a situation when you have multiple duplicate records in a table. While fetching such records, it makes more sense to fetch only unique records instead of fetching duplicate records.

Syntax:

The basic syntax of DISTINCT keyword to eliminate duplicate records is as follows:

```
SELECT DISTINCT column1, column2,. ...columnN
FROM table_name
WHERE [condition]
```

Example:

Consider the CUSTOMERS table having the following records:

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

First, let us see how the following SELECT query returns duplicate salary records:

```
SQL> SELECT SALARY FROM CUSTOMERS
     ORDER BY SALARY;
```

This would produce the following result where salary 2000 is coming twice which is a duplicate record from the original table.

```
+--------------+
| SALARY       |
+--------------+
|  1500.00 |
|  2000.00 |
|  2000.00 |
|  4500.00 |
|  6500.00 |
|  8500.00 |
| 10000.00 |
+--------------+
```

Now, let us use DISTINCT keyword with the above SELECT query and see the result:

```
SQL> SELECT DISTINCT SALARY FROM CUSTOMERS       ORDER BY SALARY;
```

This would produce the following result where we do not have any duplicate entry:

```
+--------------+
| SALARY       |
+--------------+
|  1500.00 |
|  2000.00 |
|  4500.00 |
|  6500.00 |
|  8500.00 |
| 10000.00 |
+--------------+
```

CHAPTER

SQL SORTING Results

The SQL **ORDER BY** clause is used to sort the data in ascending or descending order, based on one or more columns. Some databases sort query results in ascending order by default.

Syntax: The basic syntax of ORDER BY clause which would be used to sort result in ascending or descending order is as follows:

```
SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

You can use more than one column in the ORDER BY clause. Make sure whatever column you are using to sort, that column should be in column-list.

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 sort the result in ascending order by NAME and SALARY:

```
SQL> SELECT * FROM CUSTOMERS
     ORDER BY NAME, SALARY;
```

This would produce the following result:

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

Following is an example, which would sort the result in descending order by NAME:

```
SQL> SELECT * FROM
CUSTOMERS      ORDER BY
NAME DESC;
```

This would produce the following result:

```
+-----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
```

```
+------+-------------+-------+--------------+-------------+
|  1   | Ramesh      |  32   | Ahmedabad    |  2000.00    |
|  7   | Muffy       |  24   | Indore       | 10000.00    |
|  6   | Komal       |  22   | MP           |  4500.00    |
|  2   | Khilan      |  25   | Delhi        |  1500.00    |
|  3   | kaushik     |  23   | Kota         |  2000.00    |
|  5   | Hardik      |  27   | Bhopal       |  8500.00    |
|  4   | Chaitali    |  25   | Mumbai       |  6500.00    |
+------+-------------+-------+--------------+-------------+
```

To fetch the rows with own preferred order, the SELECT query would be as follows:

```
SQL> SELECT * FROM CUSTOMERS
    ORDER BY (CASE ADDRESS
    WHEN 'DELHI'           THEN 1
    WHEN 'BHOPAL'          THEN 2
    WHEN 'KOTA'    THEN 3
    WHEN 'AHMADABAD' THEN 4
    WHEN 'MP'   THEN 5
    ELSE 100 END) ASC, ADDRESS DESC;
```

This would produce the following result:

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

This will sort customers by ADDRESS in your ownoOrder of preference first and in a natural order for the remaining addresses. Also remaining Addresses will be sorted in the reverse alpha order.

CHAPTER

25

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.

Contraints could be column level or table level. Column level constraints are applied only to one column where as 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 chapter but its worth to revise them at this point.

Following are commonly used constraints available in SQL:

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

NOT NULL Constraint:

By default, a column can hold NULL values. If you do not want a column to have a NULL value, then you need to define such constraint on this column specifying that NULL is now not allowed for that column.

A NULL is not the same as no data, rather, it represents unknown data.

Example:

For example, the following SQL creates a new table called CUSTOMERS and adds five columns, three of which, ID and NAME and AGE, specify not to accept NULLs:

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

If CUSTOMERS table has already been created, then to add a NOT NULL constraint to SALARY column in Oracle and MySQL, you would write a statement similar to the following:

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

DEFAULT Constraint:

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

Example:

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

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

If CUSTOMERS table has already been created, then to add a DFAULT constraint to SALARY column, you would write a statement similar to the following:

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

Drop Default Constraint:

To drop a DEFAULT constraint, use the following SQL:

```
ALTER TABLE CUSTOMERS

   ALTER COLUMN SALARY DROP DEFAULT;
```

UNIQUE Constraint:

The UNIQUE Constraint prevents two records from having identical values in a particular column. In the CUSTOMERS table, for example, you might want to prevent two or more people from having identical age.

Example:

For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, AGE column is set to UNIQUE, so that you can not have two records with same age:

```
CREATE TABLE CUSTOMERS(

       ID   INT              NOT NULL,

       NAME VARCHAR (20)     NOT NULL,

       AGE  INT              NOT NULL UNIQUE,

       ADDRESS  CHAR (25) ,

       SALARY   DECIMAL (18, 2),

       PRIMARY KEY (ID)

);
```

If CUSTOMERS table has already been created, then to add a UNIQUE constraint to AGE column, you would write a statement similar to the following:

```
ALTER TABLE CUSTOMERS

   MODIFY AGE INT NOT NULL UNIQUE;
```

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

```
ALTER TABLE CUSTOMERS

   ADD CONSTRAINT myUniqueConstraint UNIQUE(AGE, SALARY);
```

DROP a UNIQUE Constraint:

To drop a UNIQUE constraint, use the following SQL:

```
ALTER TABLE CUSTOMERS

   DROP CONSTRAINT myUniqueConstraint;
```

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

```
ALTER TABLE CUSTOMERS
```

```
   DROP INDEX myUniqueConstraint;
```

PRIMARY Key:

A primary key is a field in a table which uniquely identifies each row/record in a database table. Primary keys must contain unique values. A primary key column cannot have NULL values.

A table can have only one primary key, which may consist of single or multiple fields. When multiple fields are used as a primary key, they are called a **composite key**.

If a table has a primary key defined on any field(s), then you can not have two records having the same value of that field(s).

Note: You would use these concepts while creating database tables.

Create Primary Key:

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

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

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

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

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

For defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

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

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

```
ALTER TABLE CUSTOMERS

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

Delete Primary Key:

You can clear the primary key constraints from the table, Use Syntax:

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

FOREIGN Key:

A foreign key is a key used to link two tables together. This is sometimes called a referencing key.

Primary key field from one table and insert it into the other table where it becomes a foreign key i.e., Foreign Key is a column or a combination of columns, whose values match a Primary Key in a different table.

The relationship between 2 tables matches the Primary Key in one of the tables with a Foreign Key in the second table.

If a table has a primary key defined on any field(s), then you can not have two records having the same value of that field(s).

Example:

Consider the structure of the two tables as follows:

CUSTOMERS table:

```
CREATE TABLE CUSTOMERS(

       ID   INT              NOT NULL,

       NAME VARCHAR (20)     NOT NULL,

       AGE  INT              NOT NULL,

       ADDRESS  CHAR (25) ,

       SALARY   DECIMAL (18, 2),

       PRIMARY KEY (ID)

);
```

ORDERS table:

```
CREATE TABLE ORDERS (

       ID          INT        NOT NULL,

       DATE        DATETIME,

       CUSTOMER_ID INT references CUSTOMERS(ID),
       AMOUNT     double,

       PRIMARY KEY (ID)

);
```

If ORDERS table has already been created, and the foreign key has not yet been, use the syntax for specifying a foreign key by altering a table.

```
ALTER TABLE ORDERS

   ADD FOREIGN KEY (Customer_ID) REFERENCES CUSTOMERS (ID);
```

DROP a FOREIGN KEY Constraint:

To drop a FOREIGN KEY constraint, use the following SQL:

```
ALTER TABLE ORDERS

   DROP FOREIGN KEY;
```

CHECK Constraint:

The CHECK Constraint enables a condition to check the value being entered into a record. If the condition evaluates to false, the record violates the constraint and isn't entered into the table.

Example:

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

```
CREATE TABLE CUSTOMERS(

       ID   INT              NOT NULL,

       NAME VARCHAR (20)     NOT NULL,

       AGE  INT              NOT NULL CHECK (AGE >= 18),

       ADDRESS  CHAR (25) ,

       SALARY   DECIMAL (18, 2),

       PRIMARY KEY (ID)

);
```

If CUSTOMERS table has already been created, then to add a CHECK constraint to AGE column, you would write a statement similar to the following:

```
ALTER TABLE CUSTOMERS

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

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

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

DROP a CHECK Constraint:

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

```
ALTER TABLE CUSTOMERS

   DROP CONSTRAINT myCheckConstraint;
```

INDEX:

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

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

Example:

For example, the following SQL creates a new table called CUSTOMERS and adds five columns:

```
CREATE TABLE CUSTOMERS(

       ID   INT              NOT NULL,

       NAME VARCHAR (20)     NOT NULL,

       AGE  INT              NOT NULL,

       ADDRESS  CHAR (25) ,

       SALARY   DECIMAL (18, 2),

       PRIMARY KEY (ID)

);
```

Now, you can create index on single or multiple columns using the followwng syntax:

```
CREATE INDEX index_name

    ON table_name ( column1, column2. ...);
```

To create an INDEX on AGE column, to optimize the search on customers for a particular age, following is the SQL syntax:

```
CREATE INDEX idx_age

    ON CUSTOMERS ( AGE );
```

DROP an INDEX Constraint:

To drop an INDEX constraint, use the following SQL:

```
ALTER TABLE CUSTOMERS

   DROP INDEX idx_age;
```

Constraints can be specified when a table is created with the CREATE TABLE statement or you can use ALTER TABLE statment 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.

CHAPTER

26

SQL 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, (a) CUSTOMERS table is as follows:

```
+----+----------+-----+-----------+----------+
| 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 |
+    +          +     +            +          +
```

(b) Another table is ORDERS as follows:

```
+-------+---------------------------+-----------------+-----------+
|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 follows:

```
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 |
|  3 | kaushik  |  23 |   1500 |
|  2 | Khilan   |  25 |   1560 |
|  4 | Chaitali |  25 |   2060 |
+    +          +     +        +
```

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 symbol.

SQL Join Types:

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.

INNER JOIN

The most frequently used and important 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 **INNER JOIN** is as follows:

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

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows:

```
+------+-------------+-------+-------------+-------------+
| 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    |
+------+-------------+-------+-------------+-------------+
```

(b) Another table is ORDERS as follows:

```
+-------+---------------------------+-----------------+-----------+
| OID   | DATE                      |           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
     INNER JOIN ORDERS
     ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:

```
+-----+-------------+-----------+---------------------------+
| ID  | NAME        | AMOUNT    | DATE                      |
+-----+-------------+-----------+---------------------------+
|  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       |
+-----+-------------+-----------+---------------------------+
```

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 right table, the join will still return a row in the result, but with NULL in each column from 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 **LEFT JOIN** is as follows:

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

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

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows:

```
+------+----------+-------+-----------+----------+
| 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 |
+------+----------+-------+-----------+----------+
```

(b) Another table is ORDERS as follows:

```
+-------+---------------------+-------------+--------+
| 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 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                |
+------+----------+--------+---------------------+
```

```
|  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                |
+----+----------+--------+---------------------+
```

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 left table, the join will still return a row in the result, but with NULL in each column from 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 **RIGHT JOIN** is as follows:

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

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows: +----+----------+-----+---------+----------+

```
| 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 |
+----+----------+-----+-----------+----------+
```

(b) Another table is ORDERS as follows:

```
+-----+---------------------+-------------+--------+
|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 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     |    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       |
+--------+-------------+-----------+---------------------------+
```

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 tables, and fill in NULLs for missing matches on either side.

Syntax:

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

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

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

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows:

```
+-----+-------------+-------+-------------+-------------+
| 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   |
+-----+-------------+-------+-------------+-------------+
```

(b) Another table is ORDERS as follows:

```
+-------+---------------------------+-----------------+-----------+
|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 like MySQL does not support FULL JOIN, then you can use **UNION ALL** clause to combine two JOINS as follows:

```
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
```

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_filed = b.common_field;
```

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

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows:

```
+-----+----------+-------+-----------+----------+
| 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 |
+-----+----------+-----------+
```

CARTESIAN JOIN

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

Syntax:

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

```
SELECT table1.column1, table2.column2... FROM
table1, table2 [, table3 ]
```

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows:

```
+------+----------+-------+-----------+----------+
| 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 |
+------+----------+-------+-----------+----------+
```

(b) Another table is ORDERS as follows:

```
+-------+---------------------+-------------+--------+
|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   |   3000 | 2009-10-08 00:00:00 |
|  1   | Ramesh   |   1500 | 2009-10-08 00:00:00 |
|  1   | Ramesh   |   1560 | 2009-11-20 00:00:00 |
|  1   | Ramesh   |   2060 | 2008-05-20 00:00:00 |
|  2   | Khilan   |   3000 | 2009-10-08 00:00:00 |
|  2   | Khilan   |   1500 | 2009-10-08 00:00:00 |
|  2   | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  2   | Khilan   |   2060 | 2008-05-20 00:00:00 |
|  3   | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3   | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  3   | kaushik  |   1560 | 2009-11-20 00:00:00 |
|  3   | kaushik  |   2060 | 2008-05-20 00:00:00 |
|  4   | Chaitali |   3000 | 2009-10-08 00:00:00 |
|  4   | Chaitali |   1500 | 2009-10-08 00:00:00 |
|  4   | Chaitali |   1560 | 2009-11-20 00:00:00 |
|  4   | Chaitali |   2060 | 2008-05-20 00:00:00 |
|  5   | Hardik   |   3000 | 2009-10-08 00:00:00 |
|  5   | Hardik   |   1500 | 2009-10-08 00:00:00 |
|  5   | Hardik   |   1560 | 2009-11-20 00:00:00 |
|  5   | Hardik   |   2060 | 2008-05-20 00:00:00 |
|  6   | Komal    |   3000 | 2009-10-08 00:00:00 |
|  6   | Komal    |   1500 | 2009-10-08 00:00:00 |
|  6   | Komal    |   1560 | 2009-11-20 00:00:00 |
```

```
|   6 | Komal     |   2060 | 2008-05-20 00:00:00 |
|   7 | Muffy     |   3000 | 2009-10-08 00:00:00 |
|   7 | Muffy     |   1500 | 2009-10-08 00:00:00 |
|   7 | Muffy     |   1560 | 2009-11-20 00:00:00 |
|   7 | Muffy     |   2060 | 2008-05-20 00:00:00 |
+-----+-------------+----------+---------------------------+
```

CHAPTER

SQL Unions Clause

The SQL **UNION** clause/operator is used to combine the results of two or more SELECT statements without returning any duplicate rows.

To use UNION, each SELECT must have the same number of columns selected, the same number of column expressions, the same data type, and have them in the same order, but they do not have to be the same length.

Syntax:

The basic syntax of **UNION** is as follows:

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]

UNION

SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

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

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows:

```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

(b) Another table is ORDERS as follows:

```
+-------+---------------------+-------------+--------+
|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 follows:

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
     FROM CUSTOMERS
     LEFT JOIN ORDERS
     ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION
     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                |
+--------+-------------+-----------+---------------------+
|      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                |
+--------+-------------+-----------+---------------------+
```

The UNION ALL Clause:

The UNION ALL operator is used to combine the results of two SELECT statements including duplicate rows.

The same rules that apply to UNION apply to the UNION ALL operator.

Syntax:

The basic syntax of **UNION ALL** is as follows:

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]

UNION ALL

SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

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

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows:

```
+-----+----------+-------+-----------+----------+
| 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 |
+-----+----------+-------+-----------+----------+
```

(b) Another table is ORDERS as follows:

```
+-------+---------------------------+-----------------+-----------+
|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 follows:

```
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;
```

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 |
+------+----------+--------+---------------------+
```

There are two other clauses (i.e., operators), which are very similar to UNION clause SQL INTERSECT Clause: is used to combine two SELECT statements, but returns rows only from the first SELECT statement that are identical to a row in the second SELECT statement.

- SQL EXCEPT Clause : combines two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement.

INTERSECT Clause

The SQL **INTERSECT** clause/operator is used to combine two SELECT statements, but returns rows only from the first SELECT statement that are identical to a row in the second SELECT statement. This means INTERSECT returns only common rows returned by the two SELECT statements.

Just as with the UNION operator, the same rules apply when using the INTERSECT operator. MySQL does not support INTERSECT operator

Syntax:

The basic syntax of **INTERSECT** is as follows:

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]

INTERSECT

SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

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

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows:

```
+-----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

(b) Another table is ORDERS as follows:

```
+-------+---------------------+-------------+--------+
|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 follows:

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
     FROM CUSTOMERS
     LEFT JOIN ORDERS
     ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
INTERSECT
     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    |    3000   | 2009-10-08 00:00:00 |
|      3 | kaushik    |    1500   | 2009-10-08 00:00:00 |
|      2 | Ramesh     |    1560   | 2009-11-20 00:00:00 |
|      4 | kaushik    |    2060   | 2008-05-20 00:00:00 |
+--------+------------+-----------+---------------------+
```

EXCEPT Clause

The SQL **EXCEPT** clause/operator is used to combine two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement. This means EXCEPT returns only rows, which are not available in second SELECT statement.

Just as with the UNION operator, the same rules apply when using the EXCEPT operator. MySQL does not support EXCEPT operator.

Syntax:

The basic syntax of **EXCEPT** is as follows:

```
SELECT column1 [, column2 ]

FROM table1 [, table2 ]
```

```
[WHERE condition]


EXCEPT


SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

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

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows: +----+----------+-----+---------+----------+

```
| 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 |
+    +          +     +           +          +
```

(b) Another table is ORDERS as follows:

```
+     +                     +             +        +
|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 follows:

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
```

```
     FROM CUSTOMERS
     LEFT JOIN ORDERS
     ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
EXCEPT
     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                  |
+    +          +        +                       +
|  1 | Ramesh   |   NULL | NULL                  |
|  5 | Hardik   |   NULL | NULL                  |
|  6 | Komal    |   NULL | NULL                  |
|  7 | Muffy    |   NULL | NULL                  | +----+---------+--------+------------
         +
```

CHAPTER

28

SQL NULL Values

The SQL **NULL** is the term used to represent a missing value. A NULL value in a table is a value in a field that appears to be blank.

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.

Syntax:

The basic syntax of **NULL** while creating a 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)
);
```

Here, **NOT NULL** signifies that column should always accept an explicit value of the given data type. There are two columns where we did not use NOT NULL, which means these columns could be NULL.

A field with a NULL value is one that has been left blank during record creation.

Example:

The NULL value can cause problems when selecting data, however, because when comparing an unknown value to any other value, the result is always unknown and not included in the final results.

You must use the **IS NULL** or **IS NOT NULL** operators in order to check for a NULL value.

Consider the following table, CUSTOMERS 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        |          |
|  7 | Muffy    |  24 | Indore    |          |
+----+----------+-----+-----------+----------+
```

Now, following is the usage of **IS NOT NULL** operator:

```
SQL> SELECT  ID, NAME, AGE, ADDRESS, SALARY
     FROM CUSTOMERS      WHERE SALARY IS NOT NULL;
```

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 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
+----+----------+-----+-----------+----------+
```

Now, following is the usage of **IS NULL** operator:

```
SQL> SELECT  ID, NAME, AGE, ADDRESS, SALARY
     FROM CUSTOMERS
     WHERE SALARY IS NULL;
```

This would produce the following result:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  6 | Komal    |  22 | MP        |          |
|  7 | Muffy    |  24 | Indore    |          |
+----+----------+-----+-----------+----------+
```

CHAPTER

29

SQL Alias Syntax

You can rename a table or a column temporarily by giving another name known as alias.

The use of table aliases means to rename a table in a particular SQL statement. The renaming is a temporary change and the actual table name does not change in the database.

The column aliases are used to rename a table's columns for the purpose of a particular SQL query.

Syntax:

The basic syntax of **table** alias is as follows:

```
SELECT column1, column2....
FROM table_name AS alias_name
WHERE [condition];
```

The basic syntax of **column** alias is as follows:

```
SELECT column_name AS alias_name
FROM table_name
WHERE [condition];
```

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows:

```
+------+--------------+-------+---------------+--------------+
| 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 |
+    +          +     +            +          +
```

(b) Another table is ORDERS as follows:

```
+-------+---------------------------+-----------------+-----------+
|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, following is the usage of **table alias**:

```
SQL> SELECT C.ID, C.NAME, C.AGE, O.AMOUNT
        FROM CUSTOMERS AS C, ORDERS AS O         WHERE  C.ID =
O.CUSTOMER_ID;
```

This would produce the following result:

```
+-----+-------------+-------+-----------+
| ID | NAME      | AGE | AMOUNT |
+-----+-------------+-------+-----------+
|  3 | kaushik   |  23 |   3000 |
|  3 | kaushik   |  23 |   1500 |
|  2 | Khilan    |  25 |   1560 |
|  4 | Chaitali |  25 |   2060 |
+-----+-------------+-------+-----------+
```

Following is the usage of **column alias**:

```
SQL> SELECT  ID AS CUSTOMER_ID, NAME AS CUSTOMER_NAME
     FROM CUSTOMERS      WHERE SALARY IS NOT NULL;
```

This would produce the following result:

```
+-----------------+---------------------+
| CUSTOMER_ID | CUSTOMER_NAME |
+-----------------+---------------------+
|           1 | Ramesh         |
|           2 | Khilan         |
|           3 | kaushik        |
|           4 | Chaitali       |
|           5 | Hardik         |
|           6 | Komal          |
|           7 | Muffy          |
+-----------------+---------------------+
```

CHAPTER

30

SQL Indexes

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 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 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.

CHAPTER

31

SQL ALTER TABLE Command

The SQL **ALTER TABLE** command is used to add, delete or modify columns in an existing table.

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

Syntax:

The basic syntax of **ALTER TABLE** to add a new column in an existing table is as follows:

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

The basic syntax of ALTER TABLE to **DROP COLUMN** in an existing table is as follows:

```
ALTER TABLE table_name DROP COLUMN column_name;
```

The basic syntax of ALTER TABLE to change the **DATA TYPE** of a column in a table is as follows:

```
ALTER TABLE table_name MODIFY COLUMN column_name datatype;
```

The basic syntax of ALTER TABLE to add a **NOT NULL** constraint to a column in a table is as follows:

```
ALTER TABLE table_name MODIFY column_name datatype NOT NULL;
```

The basic syntax of ALTER TABLE to **ADD UNIQUE CONSTRAINT** to a table is as follows:

```
ALTER TABLE table_name
ADD CONSTRAINT MyUniqueConstraint UNIQUE(column1, column2...);
```

The basic syntax of ALTER TABLE to **ADD CHECK CONSTRAINT** to a table is as follows:

```
ALTER TABLE table_name
ADD CONSTRAINT MyUniqueConstraint CHECK (CONDITION);
```

The basic syntax of ALTER TABLE to **ADD PRIMARY KEY** constraint to a table is as follows:

```
ALTER TABLE table_name
ADD CONSTRAINT MyPrimaryKey PRIMARY KEY (column1, column2...);
```

The basic syntax of ALTER TABLE to **DROP CONSTRAINT** from a table is as follows:

```
ALTER TABLE table_name
DROP CONSTRAINT MyUniqueConstraint;
```

If you're using MySQL, the code is as follows:

```
ALTER TABLE table_name
DROP INDEX MyUniqueConstraint;
```

The basic syntax of ALTER TABLE to **DROP PRIMARY KEY** constraint from a table is as follows:

```
ALTER TABLE table_name
DROP CONSTRAINT MyPrimaryKey;
```

If you're using MySQL, the code is as follows:

```
ALTER TABLE table_name  DROP PRIMARY KEY;
```

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 to ADD a new column in an existing table:

```
ALTER TABLE CUSTOMERS ADD SEX char(1);
```

Now, CUSTOMERS table is changed and following would be output from SELECT statement:

```
+-----+----------+-------+-----------+----------+--------+
| ID  | NAME     | AGE   | ADDRESS   | SALARY   | SEX    |
+-----+----------+-------+-----------+----------+--------+
|   1 | Ramesh   |    32 | Ahmedabad |  2000.00 | NULL   |
|   2 | Ramesh   |    25 | Delhi     |  1500.00 | NULL   |
|   3 | kaushik  |    23 | Kota      |  2000.00 | NULL   |
|   4 | kaushik  |    25 | Mumbai    |  6500.00 | NULL   |
```

```
|  5 | Hardik   |  27 | Bhopal     |  8500.00 | NULL |
|  6 | Komal    |  22 | MP         |  4500.00 | NULL |
|  7 | Muffy    |  24 | Indore     | 10000.00 | NULL |
+    +          +     +            +          +      +
```

Following is the example to DROP sex column from existing table:

```
ALTER TABLE CUSTOMERS DROP SEX;
```

Now, CUSTOMERS table is changed and following would be output from SELECT statement:

```
+------+------------+-------+--------------+-------------+
| 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 |
+------+------------+-------+--------------+-------------+
```

CHAPTER

32

SQL TRUNCATE TABLE

The SQL **TRUNCATE TABLE** command is used to delete complete data from an existing table.

You can also use DROP TABLE command to delete complete table but it would remove complete table structure form the database and you would need to re-create this table once again if you wish you store some data.

Syntax:

The basic syntax of **TRUNCATE TABLE** is as follows:

```
TRUNCATE TABLE  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 |
+----+----------+-----+-----------+----------+
```

Following is the example to truncate:

```
SQL > TRUNCATE TABLE CUSTOMERS;
```

Now, CUSTOMERS table is truncated and following would be the output from SELECT statement:

```
SQL> SELECT * FROM CUSTOMERS;
Empty set (0.00 sec)
```

CHAPTER

33

SQL - Using Views

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.

A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

Views, which are kind of virtual tables, allow users to do the following:

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data such that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.

Creating Views:

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables, or another view.

To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic CREATE VIEW syntax is as follows:

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

You can include multiple tables in your SELECT statement in very similar way as you use them in normal SQL SELECT query.

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, following is the example to create a view from CUSTOMERS table. This view would be used to have customer name and age from CUSTOMERS table:

```
SQL > CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM  CUSTOMERS;
```

Now, you can query CUSTOMERS_VIEW in similar way as you query an actual table. Following is the example:

```
SQL > SELECT * FROM CUSTOMERS_VIEW;
```

This would produce the following result:

```
+--------------+-------+
| name         | age |
+--------------+-------+
| Ramesh    |  32 |
| Khilan    |  25 |
| kaushik   |  23 |
| Chaitali  |  25 |
| Hardik    |  27 |
| Komal     |  22 |
| Muffy     |  24 |
+--------------+-------+
```

The WITH CHECK OPTION:

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition.

If they do not satisfy the condition(s), the UPDATE or INSERT returns an error.

The following is an example of creating same view CUSTOMERS_VIEW with the WITH CHECK OPTION:

```
CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM  CUSTOMERS
WHERE age IS NOT NULL
```

```
WITH CHECK OPTION;
```

The WITH CHECK OPTION in this case should deny the entry of any NULL values in the view's AGE column, because the view is defined by data that does not have a NULL value in the AGE column.

Updating a View:

A view can be updated under certain conditions:

- The SELECT clause may not contain the keyword DISTINCT.
- The SELECT clause may not contain summary functions.
- The SELECT clause may not contain set functions.
- The SELECT clause may not contain set operators.
- The SELECT clause may not contain an ORDER BY clause.
- The FROM clause may not contain multiple tables.
- The WHERE clause may not contain subqueries.
- The query may not contain GROUP BY or HAVING.
- Calculated columns may not be updated.
- All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

So if a view satisfies all the abovementioned rules then you can update a view. Following is an example to update the age of Ramesh:

```
SQL > UPDATE CUSTOMERS_VIEW
      SET AGE = 35
      WHERE name='Ramesh';
```

This would ultimately update the base table CUSTOMERS and same would reflect in the view itself. Now, try to query base table, and SELECT statement would produce the following result:

```
+------+--------------+-------+--------------+-------------+
| 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 |
+------+--------------+-------+--------------+-------------+
```

Inserting Rows into a View:

Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the INSERT command.

Here, we can not insert rows in CUSTOMERS_VIEW because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in similar way as you insert them in a table.

Deleting Rows into a View:

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.

Following is an example to delete a record having AGE= 22.

```
SQL > DELETE FROM CUSTOMERS_VIEW
WHERE age = 22;
```

This would ultimately delete a row from the base table CUSTOMERS and same would reflect in the view itself. Now, try to query base table, and SELECT statement would produce the following result:

```
+------+--------------+-------+--------------+--------------+
| 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    |
|   7  | Muffy        |   24  | Indore       |  10000.00    |
+------+--------------+-------+--------------+--------------+
```

Dropping Views:

Obviously, where you have a view, you need a way to drop the view if it is no longer needed. The syntax is very simple as given below:

```
DROP VIEW view_name;
```

Following is an example to drop CUSTOMERS_VIEW from CUSTOMERS table:

```
DROP VIEW CUSTOMERS_VIEW;
```

CHAPTER

34

SQL HAVING CLAUSE

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 *
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  |
+----+----------+-----+------------+----------+
```

CHAPTER

35

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 transaction on the table. It is important to control transactions to ensure 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 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 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 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 ];
```

CHAPTER

36

SQL Wildcard Operators

We already have discussed SQL **LIKE** operator, which is used to compare a value to similar values using wildcard operators.

SQL supports following two wildcard operators in conjunction with the LIKE operator:

Wildcards	Description
The percent sign (%)	Matches one or more characters. Note that MS Access uses the asterisk (*) wildcard character instead of the percent sign (%) wildcard character.
The underscore (_)	Matches one character. Note that MS Access uses a question mark (?) instead of the underscore (_) to match any one character.

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 | +----+----------+-----+-----------+---
----------
```

CHAPTER

37

SQL Date Functions

Following is a list of all important Date and Time related functions available through SQL. There are various other functions supported by your RDBMS. Given list is based on MySQL RDBMS.

Name	Description
ADDDATE()	Adds dates
ADDTIME()	Adds time

CONVERT_TZ()	Converts from one timezone to another
CURDATE()	Returns the current date
CURRENT_DATE(), CURRENT_DATE	Synonyms for CURDATE()
CURRENT_TIME(), CURRENT_TIME	Synonyms for CURTIME()
CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP	Synonyms for NOW()
CURTIME()	Returns the current time
DATE_ADD()	Adds two dates
DATE_FORMAT()	Formats date as specified
DATE_SUB()	Subtracts two dates
DATE()	Extracts the date part of a date or datetime expression
DATEDIFF()	Subtracts two dates
DAY()	Synonym for DAYOFMONTH()
DAYNAME()	Returns the name of the weekday
DAYOFMONTH()	Returns the day of the month (1-31)
DAYOFWEEK()	Returns the weekday index of the argument
DAYOFYEAR()	Returns the day of the year (1-366)
EXTRACT	Extracts part of a date

FROM_DAYS()	Converts a day number to a date
FROM_UNIXTIME()	Formats date as a UNIX timestamp
HOUR()	Extracts the hour
LAST_DAY	Returns the last day of the month for the argument
LOCALTIME(), LOCALTIME	Synonym for NOW()
LOCALTIMESTAMP, LOCALTIMESTAMP()	Synonym for NOW()
MAKEDATE()	Creates a date from the year and day of year
MAKETIME	MAKETIME()
MICROSECOND()	Returns the microseconds from argument
MINUTE()	Returns the minute from the argument
MONTH()	Returns the month from the date passed
MONTHNAME()	Returns the name of the month
NOW()	Returns the current date and time
PERIOD_ADD()	Adds a period to a year-month
PERIOD_DIFF()	Returns the number of months between periods
QUARTER()	Returns the quarter from a date argument
SEC_TO_TIME()	Converts seconds to 'HH:MM:SS' format

SECOND()	Returns the second (0-59)
STR_TO_DATE()	Converts a string to a date
SUBDATE()	When invoked with three arguments a synonym for DATE_SUB()
SUBTIME()	Subtracts times
SYSDATE()	Returns the time at which the function executes
TIME_FORMAT()	Formats as time
TIME_TO_SEC()	Returns the argument converted to seconds
TIME()	Extracts the time portion of the expression passed
TIMEDIFF()	Subtracts time
TIMESTAMP()	With a single argument, this function returns the date or datetime expression. With two arguments, the sum of the arguments
TIMESTAMPADD()	Adds an interval to a datetime expression
TIMESTAMPDIFF()	Subtracts an interval from a datetime expression
TO_DAYS()	Returns the date argument converted to days
UNIX_TIMESTAMP()	Returns a UNIX timestamp
UTC_DATE()	Returns the current UTC date
UTC_TIME()	Returns the current UTC time
UTC_TIMESTAMP()	Returns the current UTC date and time
WEEK()	Returns the week number

WEEKDAY()	Returns the weekday index
WEEKOFYEAR()	Returns the calendar week of the date (1-53)
YEAR()	Returns the year
YEARWEEK()	Returns the year and week

ADDDATE(date,INTERVAL expr unit), ADDDATE(expr,days)

When invoked with the INTERVAL form of the second argument, ADDDATE() is a synonym for DATE_ADD(). The related function SUBDATE() is a synonym for DATE_SUB(). For information on the INTERVAL unit argument, see the discussion for DATE_ADD().

```
mysql> SELECT DATE_ADD('1998-01-02', INTERVAL 31 DAY); +----------------------------
-----------------------------------+
| DATE_ADD('1998-01-02', INTERVAL 31 DAY)                    |
+------------------------------------------------------------------------------+
| 1998-02-02                                                 |
+------------------------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT ADDDATE('1998-01-02', INTERVAL 31 DAY); +----------------------------
-----------------------------------+
| ADDDATE('1998-01-02', INTERVAL 31 DAY)                     |
+------------------------------------------------------------------------------+
| 1998-02-02                                                 |
+------------------------------------------------------------------------------+ 1 row in set (0.00 sec)
```

When invoked with the days form of the second argument, MySQL treats it as an integer number of days to be added to expr.

```
mysql> SELECT ADDDATE('1998-01-02', 31);
+------------------------------------------------------------------------------+
| DATE_ADD('1998-01-02', INTERVAL 31 DAY)                    |
+------------------------------------------------------------------------------+
| 1998-02-02                                                 |
+------------------------------------------------------------------------------+ 1 row in set (0.00 sec)
```

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.

```
mysql> SELECT ADDTIME('1997-12-31 23:59:59.999999','1 1:1:1.000002');
+-----------------------------------------------------------------------------+
| DATE_ADD('1997-12-31 23:59:59.999999','1 1:1:1.000002') |
+-----------------------------------------------------------------------------+
| 1998-01-02 01:01:01.000001                                                  |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
mysql> SELECT CONVERT_TZ('2004-01-01 12:00:00','GMT','MET'); +----------------------
-------------------------------------------+
| CONVERT_TZ('2004-01-01 12:00:00','GMT','MET')            |
+----------------------------------------------------------------------+
| 2004-01-01 13:00:00                                                  |
+----------------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT CONVERT_TZ('2004-01-01 12:00:00','+00:00','+10:00');
+----------------------------------------------------------------------+
| CONVERT_TZ('2004-01-01 12:00:00','+00:00','+10:00')      |
+----------------------------------------------------------------------+
| 2004-01-01 22:00:00                                                  |
+----------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
mysql> SELECT CURDATE();
+----------------------------------------------------------------------+
| CURDATE()                                                            |
+----------------------------------------------------------------------+
| 1997-12-15                                                           |
+----------------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT CURDATE() + 0;
+----------------------------------------------------------------------+
| CURDATE() + 0                                                        |
+----------------------------------------------------------------------+
| 19971215                                                             |
+----------------------------------------------------------------------+ 1 row in set (0.00 sec)
```

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.

```
mysql> SELECT CURTIME();
+----------------------------------------------------------------------+
| CURTIME()                                                            |
+----------------------------------------------------------------------+
| 23:50:26                                                             |
+----------------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT CURTIME() + 0;
+----------------------------------------------------------------------+
```

```
| CURTIME() + 0                                                                |
+------------------------------------------------------------------------------+
| 235026                                                                       |
+------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
mysql> SELECT DATE('2003-12-31 01:02:03');
+------------------------------------------------------------------------------+
| DATE('2003-12-31 01:02:03')                                                  |
+------------------------------------------------------------------------------+
|  2003-12-31                                                                  |
+------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
mysql> SELECT DATEDIFF('1997-12-31 23:59:59','1997-12-30'); +-------------------------
-----------------------------------------+
| DATEDIFF('1997-12-31 23:59:59','1997-12-30')                                 |
+------------------------------------------------------------------------------+
| 1                                                                            |
+------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

DATE_ADD(date,INTERVAL expr unit), DATE_SUB(date,INTERVAL expr unit)

These functions perform date arithmetic. date is a DATETIME or DATE value specifying the starting date. expr is an expression specifying the interval value to be added or subtracted from the starting date. expr is a string; it may start with a '-' for negative intervals. unit is a keyword indicating the units in which the expression should be interpreted.

The INTERVAL keyword and the unit specifier are not case sensitive.

The following table shows the expected form of the expr argument for each unit value;

unit **Value**	**Expected**expr**Format**
MICROSECOND	MICROSECONDS
SECOND	SECONDS
MINUTE	MINUTES
HOUR	HOURS
DAY	DAYS
WEEK	WEEKS
MONTH	MONTHS
QUARTER	QUARTERS
YEAR	YEARS
SECOND_MICROSECOND	'SECONDS.MICROSECONDS'
MINUTE_MICROSECOND	'MINUTES.MICROSECONDS'
MINUTE_SECOND	'MINUTES:SECONDS'
HOUR_MICROSECOND	'HOURS.MICROSECONDS'
HOUR_SECOND	'HOURS:MINUTES:SECONDS'
HOUR_MINUTE	'HOURS:MINUTES'

DAY_MICROSECOND	'DAYS.MICROSECONDS'
DAY_SECOND	'DAYS HOURS:MINUTES:SECONDS'
DAY_MINUTE	'DAYS HOURS:MINUTES'
DAY_HOUR	'DAYS HOURS'
YEAR_MONTH	'YEARS-MONTHS'

The values QUARTER and WEEK are available beginning with MySQL 5.0.0.

```
mysql> SELECT DATE_ADD('1997-12-31 23:59:59',
    -> INTERVAL '1:1' MINUTE_SECOND);
+                                                +
| DATE_ADD('1997-12-31 23:59:59', INTERVAL...    |
+                                                +
| 1998-01-01 00:01:00                            |
+                                                +
1 row in set (0.00 sec)

mysql> SELECT DATE_ADD('1999-01-01', INTERVAL 1 HOUR); +-----------------------------
-------------------------------------------------------------------
| DATE_ADD('1999-01-01', INTERVAL 1 HOUR)          |
+                                                  +
| 1999-01-01 01:00:00                              |
+                                                  +
1 row in set (0.00 sec)
```

DATE_FORMAT(date,format)

Formats the date value according to the format string.

The following specifiers may be used in the format string. The '%' character is required before format specifier characters.

Specifier	Description
%a	Abbreviated weekday name (Sun..Sat)
%b	Abbreviated month name (Jan..Dec)
%c	Month, numeric (0..12)
%D	Day of the month with English suffix (0th, 1st, 2nd, 3rd, .)

%d	Day of the month, numeric (00..31)
%e	Day of the month, numeric (0..31)
%f	Microseconds (000000..999999)
%H	Hour (00..23)
%h	Hour (01..12)
%I	Hour (01..12)
%i	Minutes, numeric (00..59)
%j	Day of year (001..366)
%k	Hour (0..23)
%l	Hour (1..12)
%M	Month name (January..December)
%m	Month, numeric (00..12)
%p	AM or PM
%r	Time, 12-hour (hh:mm:ss followed by AM or PM)
%S	Seconds (00..59)
%s	Seconds (00..59)
%T	Time, 24-hour (hh:mm:ss)
%U	Week (00..53), where Sunday is the first day of the week

%u	Week (00..53), where Monday is the first day of the week
%V	Week (01..53), where Sunday is the first day of the week; used with %X
%v	Week (01..53), where Monday is the first day of the week; used with %x
%W	Weekday name (Sunday..Saturday)
%w	Day of the week (0=Sunday..6=Saturday)
%X	Year for the week where Sunday is the first day of the week, numeric, four digits; used with %V
%x	Year for the week, where Monday is the first day of the week, numeric, four digits; used with %v
%Y	Year, numeric, four digits
%y	Year, numeric (two digits)
%%	A literal .%. character
%x	x, for any.x. not listed above

```
mysql> SELECT DATE_FORMAT('1997-10-04 22:23:00', '%W %M %Y');+----------------------
----------------------------------------------------------------------------------------
| DATE_FORMAT('1997-10-04 22:23:00', '%W %M %Y')          |
+                                                        +
| Saturday October 1997                                  |
+                                                        +
1 row in set (0.00 sec)

mysql> SELECT DATE_FORMAT('1997-10-04 22:23:00'
    -> '%H %k %I %r %T %S %w');
+                                                        +
| DATE_FORMAT('1997-10-04 22:23:00.......                |
+                                                        +
|  22 22 10 10:23:00 PM 22:23:00 00 6                    |
+                                                        +
 1 row in set (0.00 sec)
```

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.

```
mysql> SELECT DAYNAME('1998-02-05');
+-----------------------------------------------------------------------------+
| DAYNAME('1998-02-05')                                                       |
+-----------------------------------------------------------------------------+
| Thursday                                                                    |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

DAYOFMONTH(date)

Returns the day of the month for date, in the range 0 to 31.

```
mysql> SELECT DAYOFMONTH('1998-02-03');
+-----------------------------------------------------------------------------+
| DAYOFMONTH('1998-02-03')                                                    |
+-----------------------------------------------------------------------------+
| 3                                                                           |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

DAYOFWEEK(date)

Returns the weekday index for date (1 = Sunday, 2 = Monday, ., 7 = Saturday). These index values correspond to the ODBC standard.

```
mysql> SELECT DAYOFWEEK('1998-02-03');
+-----------------------------------------------------------------------------+
|DAYOFWEEK('1998-02-03')                                                      |
+-----------------------------------------------------------------------------+
| 3                                                                           |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

DAYOFYEAR(date)

Returns the day of the year for date, in the range 1 to 366.

```
mysql> SELECT DAYOFYEAR('1998-02-03');
+-----------------------------------------------------------------------------+
| DAYOFYEAR('1998-02-03')                                                     |
```

```
+---------------------------------------------------------------------------+
| 34                                                                        |
+---------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
mysql> SELECT EXTRACT(YEAR FROM '1999-07-02');
+---------------------------------------------------------------------------+
| EXTRACT(YEAR FROM '1999-07-02')                                           |
+---------------------------------------------------------------------------+
| 1999                                                                      |
+---------------------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT EXTRACT(YEAR_MONTH FROM '1999-07-02 01:02:03');+----------------------
-------------------------------------------+
| EXTRACT(YEAR_MONTH FROM '1999-07-02 01:02:03')                            |
+---------------------------------------------------------------------------+
| 199907                                                                    |
+---------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

FROM_DAYS(N)

Given a day number N, returns a DATE value.

```
mysql> SELECT FROM_DAYS(729669);
+---------------------------------------------------------------------------+
| FROM_DAYS(729669)                                                         |
+---------------------------------------------------------------------------+
| 1997-10-07                                                                |
+---------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

Use FROM_DAYS() with caution on old dates. It is not intended for use with values that precede the advent of the Gregorian calendar (1582).

FROM_UNIXTIME(unix_timestamp) FROM_UNIXTIME(unix_timestamp,for mat)

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. The value is expressed in the current time zone. unix_timestamp is an internal timestamp value such as is produced by the UNIX_TIMESTAMP() function.

If format is given, the result is formatted according to the format string, which is used the same way as listed in the entry for the DATE_FORMAT() function.

```
mysql> SELECT FROM_UNIXTIME(875996580); +-----------------------------------------------------------
-----------------+
```

```
| FROM_UNIXTIME(875996580)                                                |
+-------------------------------------------------------------------------+
| 1997-10-04 22:23:00                                                     |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

HOUR(time)

Returns the hour for time. The range of the return value is 0 to 23 for time-of-day values. However, the range of TIME values actually is much larger, so HOUR can return values greater than 23.

```
mysql> SELECT HOUR('10:05:03');
+-------------------------------------------------------------------------+
| HOUR('10:05:03')                                                        |
+-------------------------------------------------------------------------+
| 10                                                                      |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
mysql> SELECT LAST_DAY('2003-02-05');
+-------------------------------------------------------------------------+
| LAST_DAY('2003-02-05')                                                  |
+-------------------------------------------------------------------------+
| 2003-02-28                                                              |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
mysql> SELECT MAKEDATE(2001,31), MAKEDATE(2001,32); +--------------------------------
---------------------------------+
| MAKEDATE(2001,31), MAKEDATE(2001,32)                                    |
+-------------------------------------------------------------------------+
| '2001-01-31', '2001-02-01'                                              |
+-------------------------------------------------------------------------+ 1 row in set (0.00 sec)
```

MAKETIME(hour,minute,second)

Returns a time value calculated from the hour, minute and second arguments.

```
mysql> SELECT MAKETIME(12,15,30);
+-----------------------------------------------------------------------------+
| MAKETIME(12,15,30)                                                          |
+-----------------------------------------------------------------------------+
| '12:15:30'                                                                  |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

MICROSECOND(expr)

Returns the microseconds from the time or datetime expression expr as a number in the range from 0 to 999999.

```
mysql> SELECT MICROSECOND('12:00:00.123456');
+-----------------------------------------------------------------------------+
| MICROSECOND('12:00:00.123456')                                              |
+-----------------------------------------------------------------------------+
| 123456                                                                      |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

MINUTE(time)

Returns the minute for time, in the range 0 to 59.

```
mysql> SELECT MINUTE('98-02-03 10:05:03');
+-----------------------------------------------------------------------------+
| MINUTE('98-02-03 10:05:03')                                                 |
+-----------------------------------------------------------------------------+
| 5                                                                           |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

MONTH(date)

Returns the month for date, in the range 0 to 12.

```
mysql> SELECT MONTH('1998-02-03')
+-----------------------------------------------------------------------------+
| MONTH('1998-02-03')                                                         |
+-----------------------------------------------------------------------------+
| 2                                                                           |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

MONTHNAME(date)

Returns the full name of the month for date.

```
mysql> SELECT MONTHNAME('1998-02-05');
```

```
+------------------------------------------------------------------------+
| MONTHNAME('1998-02-05')                                                |
+------------------------------------------------------------------------+
| February                                                               |
+------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
mysql> SELECT NOW();
+------------------------------------------------------------------------+
| NOW()                                                                  |
+------------------------------------------------------------------------+
| 1997-12-15 23:50:26                                                    |
+------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
mysql> SELECT PERIOD_ADD(9801,2);
+------------------------------------------------------------------------+
| PERIOD_ADD(9801,2)                                                     |
+------------------------------------------------------------------------+
| 199803                                                                 |
+------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
mysql> SELECT PERIOD_DIFF(9802,199703);
+------------------------------------------------------------------------+
| PERIOD_DIFF(9802,199703)                                               |
+------------------------------------------------------------------------+
| 11                                                                     |
+------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

QUARTER(date)

Returns the quarter of the year for date, in the range 1 to 4.

```
mysql> SELECT QUARTER('98-04-01');
+------------------------------------------------------------------------+
```

```
| QUARTER('98-04-01')                                                    |
+------------------------------------------------------------------------+
| 2                                                                      |
+------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

SECOND(time)

Returns the second for time, in the range 0 to 59.

```
mysql> SELECT SECOND('10:05:03');
+------------------------------------------------------------------------+
| SECOND('10:05:03')                                                     |
+------------------------------------------------------------------------+
| 3                                                                      |
+------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
mysql> SELECT SEC_TO_TIME(2378);
+------------------------------------------------------------------------+
| SEC_TO_TIME(2378)                                                      |
+------------------------------------------------------------------------+
| 00:39:38                                                               |
+------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
mysql> SELECT STR_TO_DATE('04/31/2004', '%m/%d/%Y'); +-------------------------------
---------------------------------+
| STR_TO_DATE('04/31/2004', '%m/%d/%Y')                                  |
+------------------------------------------------------------------------+
| 2004-04-31                                                             |
+------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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().

```
mysql> SELECT DATE_SUB('1998-01-02', INTERVAL 31 DAY); +-------------------------------
-----------------------------------+
| DATE_SUB('1998-01-02', INTERVAL 31 DAY)                                |
```

```
+------------------------------------------------------------------------+
| 1997-12-02                                                             |
+------------------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT SUBDATE('1998-01-02', INTERVAL 31 DAY);+-------------------------------
---------------------------------+
| SUBDATE('1998-01-02', INTERVAL 31 DAY)                                 |
+------------------------------------------------------------------------+
| 1997-12-02                                                             |
+------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
mysql> SELECT SUBTIME('1997-12-31 23:59:59.999999',
    -> '1 1:1:1.000002');
+------------------------------------------------------------------------+
| SUBTIME('1997-12-31 23:59:59.999999'...                                |
+------------------------------------------------------------------------+
| 1997-12-30 22:58:58.999997                                             |
+------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
mysql> SELECT SYSDATE();
+------------------------------------------------------------------------+
| SYSDATE()                                                              |
+------------------------------------------------------------------------+
| 2006-04-12 13:47:44                                                    |
+------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

TIME(expr)

Extracts the time part of the time or datetime expression expr and returns it as a string.

```
mysql> SELECT TIME('2003-12-31 01:02:03');
+------------------------------------------------------------------------+
| TIME('2003-12-31 01:02:03')                                            |
+------------------------------------------------------------------------+
| 01:02:03                                                               |
+------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
mysql> SELECT TIMEDIFF('1997-12-31 23:59:59.000001',
    -> '1997-12-30 01:01:01.000002'); +-----------------------------------------
--------------+
| TIMEDIFF('1997-12-31 23:59:59.000001'.....              |
+-----------------------------------------------------------------------------+
|  46:58:57.999999                                        |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
mysql> SELECT TIMESTAMP('2003-12-31');
+-----------------------------------------------------------------------------+
| TIMESTAMP('2003-12-31')                                 |
+-----------------------------------------------------------------------------+
| 2003-12-31 00:00:00                                     |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

TIMESTAMPADD(unit,interval,datetime_expr)

Adds the integer expression interval to the date or datetime expression datetime_expr. The unit for interval is given by the unit argument, which should be one of the following values: FRAC_SECOND, SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, QUARTER or YEAR.

The unit value may be specified using one of keywords as shown or with a prefix of SQL_TSI_. For example, DAY and SQL_TSI_DAY both are legal.

```
mysql> SELECT TIMESTAMPADD(MINUTE,1,'2003-01-02'); +---------------------------------
--------------------------------+
| TIMESTAMPADD(MINUTE,1,'2003-01-02')                     |
+-----------------------------------------------------------------------------+
| 2003-01-02 00:01:00                                     |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

TIMESTAMPDIFF(unit,datetime_expr1,datetime_expr2)

Returns the integer difference between the date or datetime expressions datetime_expr1 and datetime_expr2. The unit for the result is given by the unit argument. The legal values for unit are the same as those listed in the description of the TIMESTAMPADD() function.

```
mysql> SELECT TIMESTAMPDIFF(MONTH,'2003-02-01','2003-05-01'); +--------------------
-----------------------------------------------+
| TIMESTAMPDIFF(MONTH,'2003-02-01','2003-05-01')          |
+-----------------------------------------------------------------------------+
```

```
| 3                                                                          |
+----------------------------------------------------------------------------+ 1 row in set (0.00 sec)
```

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.

If the time value contains an hour part that is greater than 23, the %H and %k hour format specifiers produce a value larger than the usual range of 0..23. The other hour format specifiers produce the hour value modulo 12.

```
mysql> SELECT TIME_FORMAT('100:00:00', '%H %k %h %I %l'); +--------------------------
----------------------------------------+
| TIME_FORMAT('100:00:00', '%H %k %h %I %l')                                 |
+----------------------------------------------------------------------------+
| 100 100 04 04 4                                                            |
+----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

TIME_TO_SEC(time)

Returns the time argument converted to seconds.

```
mysql> SELECT TIME_TO_SEC('22:23:00');
+----------------------------------------------------------------------------+
| TIME_TO_SEC('22:23:00')                                                    |
+----------------------------------------------------------------------------+
| 80580                                                                      |
+----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

TO_DAYS(date)

Given a date, returns a day number (the number of days since year 0).

```
mysql> SELECT TO_DAYS(950501);
+----------------------------------------------------------------------------+
| TO_DAYS(950501)                                                            |
+----------------------------------------------------------------------------+
| 728779                                                                     |
+----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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. If UNIX_TIMESTAMP() is called with a date argument, it returns the value of the argument as seconds since '1970-01-01 00:00:00' UTC. date may be a DATE string, a DATETIME string, a TIMESTAMP, or a number in the format YYMMDD or YYYYMMDD.

```
mysql> SELECT UNIX_TIMESTAMP();
+----------------------------------------------------------------------------+
| UNIX_TIMESTAMP()                                                           |
+----------------------------------------------------------------------------+
| 882226357                                                                  |
+----------------------------------------------------------------------------+
```

```
1 row in set (0.00 sec)

mysql> SELECT UNIX_TIMESTAMP('1997-10-04 22:23:00'); +------------------------------
---------------------------------+
| UNIX_TIMESTAMP('1997-10-04 22:23:00')                    |
+----------------------------------------------------------------------------+
| 875996580                                                |
+----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
mysql> SELECT UTC_DATE(), UTC_DATE() + 0;
+----------------------------------------------------------------------------+
| UTC_DATE(), UTC_DATE() + 0                                |
+----------------------------------------------------------------------------+
| 2003-08-14, 20030814                                     |
+----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
mysql> SELECT UTC_TIME(), UTC_TIME() + 0;
+----------------------------------------------------------------------------+
| UTC_TIME(), UTC_TIME() + 0                                |
+----------------------------------------------------------------------------+
| 18:07:53, 180753                                         |
+----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
mysql> SELECT UTC_TIMESTAMP(), UTC_TIMESTAMP() + 0; +--------------------------------
--------------------------------+
| UTC_TIMESTAMP(), UTC_TIMESTAMP() + 0                     |
+----------------------------------------------------------------------------+
| 2003-08-14 18:08:04, 20030814180804                      |
+----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

WEEK(date[,mode])

This function returns the week number for date. The two-argument form of WEEK() allows you to specify whether the week starts on Sunday or Monday and whether the return value should be in the range from 0 to 53 or from 1 to 53. If the mode argument is omitted, the value of the default_week_format system variable is used

Mode	First Day of week	Range	Week 1 is the first week.
0	Sunday	0-53	with a Sunday in this year
1	Monday	0-53	with more than 3 days this year
2	Sunday	1-53	with a Sunday in this year
3	Monday	1-53	with more than 3 days this year
4	Sunday	0-53	with more than 3 days this year
5	Monday	0-53	with a Monday in this year
6	Sunday	1-53	with more than 3 days this year
7	Monday	1-53	with a Monday in this year

```
mysql> SELECT WEEK('1998-02-20');
+-------------------------------------------------------------------------+
| WEEK('1998-02-20')                                                      |
+-------------------------------------------------------------------------+
| 7                                                                       |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

WEEKDAY(date)

Returns the weekday index for date (0 = Monday, 1 = Tuesday, . 6 = Sunday).

```
mysql> SELECT WEEKDAY('1998-02-03 22:23:00');
+-------------------------------------------------------------------------+
| WEEKDAY('1998-02-03 22:23:00')                                          |
+-------------------------------------------------------------------------+
| 1                                                                       |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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).

```
mysql> SELECT WEEKOFYEAR('1998-02-20');
+-------------------------------------------------------------------------+
```

```
| WEEKOFYEAR('1998-02-20')                                                |
+-------------------------------------------------------------------------+
| 8                                                                       |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

YEAR(date)

Returns the year for date, in the range 1000 to 9999, or 0 for the .zero. date.

```
mysql> SELECT YEAR('98-02-03');
+-------------------------------------------------------------------------+
| YEAR('98-02-03')                                                        |
+-------------------------------------------------------------------------+
| 1998                                                                    | +-------------------------
-----------------------------------------+
1 row in set (0.00 sec)
```

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.

```
mysql> SELECT YEARWEEK('1987-01-01');
+-------------------------------------------------------------------------------+
| YEAR('98-02-03')YEARWEEK('1987-01-01')                                        |
+-------------------------------------------------------------------------------+
| 198653                                                                        |
+-------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

Note that the week number is different from what the WEEK() function would return (0) for optional arguments 0 or 1, as WEEK() then returns the week in the context of the given year.

For more information, check MySQL Official Website - Date and Time Functions

CHAPTER

38

SQL Temporary Tables

There are RDBMS, which support temporary tables. Temporary Tables are a great feature that lets you store and process intermediate results by using the same selection, update, and join capabilities that you can use with typical SQL Server tables.

The temporary tables could be very useful in some cases to keep temporary data. The most important thing that should be known for temporary tables is that they will be deleted when the current client session terminates.

Temporary tables are available in MySQL version 3.23 onwards. If you use an older version of MySQL than 3.23, you can't use temporary tables, but you can use heap tables.

As stated earlier, temporary tables will only last as long as the session is alive. If you run the code in a PHP script, the temporary table will be destroyed automatically when the script finishes executing. If you are connected to the MySQL database server through the MySQL client program, then the temporary table will exist until you close the client or manually destroy the table.

Example:

Here is an example showing you usage of temporary table:

```
mysql> CREATE TEMPORARY TABLE SALESSUMMARY (
    -> product_name VARCHAR(50) NOT NULL
    -> , total_sales DECIMAL(12,2) NOT NULL DEFAULT 0.00
    -> , avg_unit_price DECIMAL(7,2) NOT NULL DEFAULT 0.00
-> , total_units_sold INT UNSIGNED NOT NULL DEFAULT 0 );
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO SALESSUMMARY
    -> (product_name, total_sales, avg_unit_price, total_units_sold)
-> VALUES
    -> ('cucumber', 100.25, 90, 2);

mysql> SELECT * FROM SALESSUMMARY;
+--------------+-------------+---------------+----------------+
| product_name | total_sales | avg_unit_price | total_units_sold |
+--------------+-------------+---------------+----------------+
| cucumber     |      100.25 |          90.00 |              2 |
+--------------+-------------+---------------+----------------+
1 row in set (0.00 sec)
```

When you issue a **SHOW TABLES** command, then your temporary table would not be listed out in the list. Now if you will log out of the MySQL session and then you will issue a SELECT command, then you will find no data available in the database. Even your temporary table would also not exist.

Dropping Temporary Tables:

By default, all the temporary tables are deleted by MySQL when your database connection gets terminated. Still you want to delete them in between, then you do so by issuing DROP TABLE command.

Following is the example on dropping a temproary table.

```
mysql> CREATE TEMPORARY TABLE SALESSUMMARY (
    -> product_name VARCHAR(50) NOT NULL
    -> , total_sales DECIMAL(12,2) NOT NULL DEFAULT 0.00
    -> , avg_unit_price DECIMAL(7,2) NOT NULL DEFAULT 0.00
-> , total_units_sold INT UNSIGNED NOT NULL DEFAULT 0
);
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO SALESSUMMARY
    -> (product_name, total_sales, avg_unit_price, total_units_sold)
    -> VALUES
    -> ('cucumber', 100.25, 90, 2);

mysql> SELECT * FROM SALESSUMMARY;
+--------------+-------------+---------------+----------------+
| product_name | total_sales | avg_unit_price | total_units_sold |
+--------------+-------------+---------------+----------------+
| cucumber     |      100.25 |          90.00 |              2 |
+--------------+-------------+---------------+----------------+
1 row in set (0.00 sec) mysql> DROP
TABLE SALESSUMMARY; mysql>  SELECT
* FROM SALESSUMMARY;
ERROR 1146: Table 'TUTORIALS.SALESSUMMARY' doesn't exist
```

CHAPTER

39

SQL Clone Tables

There may be a situation when you need an exact copy of a table and CREATE TABLE ... SELECT... doesn't suit your purposes because the copy must include the same indexes, default values, and so forth.

If you are using MySQL RDBMS, you can handle this situation by the following steps:

- Use SHOW CREATE TABLE command to get a CREATE TABLE statement that specifies the source table's structure, indexes and all.
- Modify the statement to change the table name to that of the clone table and execute the statement. This way you will have exact clone table.
- Optionally, if you need the table contents copied as well, issue an INSERT INTO ... SELECT statement, too.

Example:

Try out the following example to create a clone table for **TUTORIALS_TBL,** whose structure is as follows:

Step 1:

Get complete structure about table.

```
SQL> SHOW CREATE TABLE TUTORIALS_TBL \G;
*************************** 1. row ***************************
       Table: TUTORIALS_TBL
Create Table: CREATE TABLE `TUTORIALS_TBL` (
  `tutorial_id` int(11) NOT NULL auto_increment,
  `tutorial_title` varchar(100) NOT NULL default '',
  `tutorial_author` varchar(40) NOT NULL default '',
  `submission_date` date default NULL,
```

```
  PRIMARY KEY  (`tutorial_id`),
  UNIQUE KEY `AUTHOR_INDEX` (`tutorial_author`)
) TYPE=MyISAM
1 row in set (0.00 sec)
```

Step 2:

Rename this table and create another table.

```
SQL> CREATE TABLE `CLONE_TBL` (
  -> `tutorial_id` int(11) NOT NULL auto_increment,
  -> `tutorial_title` varchar(100) NOT NULL default '',
  -> `tutorial_author` varchar(40) NOT NULL default '',
  -> `submission_date` date default NULL,
  -> PRIMARY KEY  (`tutorial_id`),
  -> UNIQUE KEY `AUTHOR_INDEX` (`tutorial_author`)
-> ) TYPE=MyISAM;
Query OK, 0 rows affected (1.80 sec)
```

Step 3:

After executing step 2, you will clone a table in your database. If you want to copy data from old table, then you can do it by using INSERT INTO... SELECT statement.

```
SQL> INSERT INTO CLONE_TBL (tutorial_id,
    ->                      tutorial_title,
    ->                      tutorial_author,
    ->                      submission_date)     -> SELECT
tutorial_id,tutorial_title,
    ->        tutorial_author,submission_date,
    -> FROM TUTORIALS_TBL;
Query OK, 3 rows affected (0.07 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

Finally, you will have exact clone table as you wanted to have.

CHAPTER

40

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 the 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 |
+----+----------+-----+-----------+----------+
```

CHAPTER

41

SQL – Using Sequences

A sequence is a set of integers 1, 2, 3, ... that are generated in order on demand. Sequences are frequently used in databases because many applications require each row in a table to contain a unique value, and sequences provide an easy way to generate them.

This chapter describes how to use sequences in MySQL.

Using AUTO_INCREMENT column:

The simplest way in MySQL to use sequences is to define a column as AUTO_INCREMENT and leave rest of the things to MySQL to take care.

Example:

Try out the following example. This will create table and after that it will insert few rows in this table where it is not required to give record ID because its auto-incremented by MySQL.

```
mysql> CREATE TABLE INSECT
    -> (
    -> id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    -> PRIMARY KEY (id),
    -> name VARCHAR(30) NOT NULL, # type of insect
    -> date DATE NOT NULL, # date collected
    -> origin VARCHAR(30) NOT NULL # where collected
);
Query OK, 0 rows affected (0.02 sec)
mysql> INSERT INTO INSECT (id,name,date,origin) VALUES
    -> (NULL,'housefly','2001-09-10','kitchen'),
    -> (NULL,'millipede','2001-09-10','driveway'),
    -> (NULL,'grasshopper','2001-09-10','front yard');
Query OK, 3 rows affected (0.02 sec) Records:
3  Duplicates: 0  Warnings: 0 mysql> SELECT *
FROM INSECT ORDER BY id; +----+ ------------ +-
--------------+----------------+
| id | name        | date       | origin      |
+------+-----------------+----------------+----------------+
|  1 | housefly    | 2001-09-10 | kitchen     |
|  2 | millipede   | 2001-09-10 | driveway    |
|  3 | grasshopper | 2001-09-10 | front yard |
+------+-----------------+----------------+----------------+
3 rows in set (0.00 sec)
```

Obtain AUTO_INCREMENT Values:

LAST_INSERT_ID( ) is a SQL function, so you can use it from within any client that understands how to issue SQL statements. Otherwise, PERL and PHP scripts provide exclusive functions to retrieve auto-incremented value of last record.

PERL Example:

Use the mysql_insertid attribute to obtain the AUTO_INCREMENT value generated by a query. This attribute is accessed through either a database handle or a statement handle, depending on how you issue the query. The following example references it through the database handle:

```
$dbh->do ("INSERT INTO INSECT (name,date,origin)
VALUES('moth','2001-09-14','windowsill')"); my
$seq = $dbh->{mysql_insertid};
```

PHP Example:

After issuing a query that generates an AUTO_INCREMENT value, retrieve the value by calling mysql_insert_id( ):

```
mysql_query ("INSERT INTO INSECT (name,date,origin) VALUES('moth','2001-09-
14','windowsill')", $conn_id);
$seq = mysql_insert_id ($conn_id);
```

Renumbering an Existing Sequence:

There may be a case when you have deleted many records from a table and you want to resequence all the records. This can be done by using a simple trick but you should be very careful to do so if your table is having joins with other table.

If you determine that resequencing an AUTO_INCREMENT column is unavoidable, the way to do it is to drop the column from the table, then add it again. The following example shows how to renumber the id values in the insect table using this technique:

```
mysql> ALTER TABLE INSECT DROP id; mysql>
ALTER TABLE insect
    -> ADD id INT UNSIGNED NOT NULL AUTO_INCREMENT FIRST,
-> ADD PRIMARY KEY (id);
```

Starting a Sequence at a Particular Value:

By default, MySQL will start sequence from 1 but you can specify any other number as well at the time of table creation. Following is the example where MySQL will start sequence from 100.

```
mysql> CREATE TABLE INSECT
    -> (
    -> id INT UNSIGNED NOT NULL AUTO_INCREMENT = 100,
    -> PRIMARY KEY (id),
    -> name VARCHAR(30) NOT NULL, # type of insect
    -> date DATE NOT NULL, # date collected
    -> origin VARCHAR(30) NOT NULL # where collected );
```

Alternatively, you can create the table and then set the initial sequence value with ALTER TABLE.

```
mysql> ALTER TABLE t AUTO_INCREMENT = 100;
```

CHAPTER

42

SQL – Handling Duplicates

There may be a situation when you have multiple duplicate records in a table. While fetching such records, it makes more sense to fetch only unique records instead of fetching duplicate records.

The SQL **DISTINCT** keyword, which we already have discussed, is used in conjunction with SELECT statement to eliminate all the duplicate records and fetching only unique records.

Syntax:

The basic syntax of DISTINCT keyword to eliminate duplicate records is as follows:

```
SELECT DISTINCT column1, column2,. ...columnN
FROM table_name
WHERE [condition]
```

Example:

Consider the CUSTOMERS table having the following records:

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

First, let us see how the following SELECT query returns duplicate salary records:

```
SQL> SELECT SALARY FROM CUSTOMERS
     ORDER BY SALARY;
```

This would produce the following result where salary 2000 is coming twice which is a duplicate record from the original table.

```
+----------+
| SALARY   |
+----------+
|  1500.00 |
|  2000.00 |
|  2000.00 |
|  4500.00 |
|  6500.00 |
|  8500.00 |
| 10000.00 |
+----------+
```

Now, let us use DISTINCT keyword with the above SELECT query and see the result:

```
SQL> SELECT DISTINCT SALARY FROM CUSTOMERS      ORDER BY SALARY;
```

This would produce the following result where we do not have any duplicate entry:

```
+----------+
| SALARY   |
+----------+
|  1500.00 |
|  2000.00 |
|  4500.00 |
|  6500.00 |
|  8500.00 |
| 10000.00 |
+----------+
```

CHAPTER

43

SQL Useful Functions

SQL has many built-in functions for performing processing on string or numeric data. Following is the list of all useful SQL built-in functions:

- SQL COUNT Function - The SQL COUNT aggregate function is used to count the number of rows in a database table.
- SQL MAX Function - The SQL MAX aggregate function allows us to select the highest (maximum) value for a certain column.
- SQL MIN Function - The SQL MIN aggregate function allows us to select the lowest (minimum) value for a certain column.
- SQL AVG Function - The SQL AVG aggregate function selects the average value for certain table column.
- SQL SUM Function - The SQL SUM aggregate function allows selecting the total for a numeric column.
- SQL SQRT Functions - This is used to generate a square root of a given number.
- SQL RAND Function - This is used to generate a random number using SQL command.
- SQL CONCAT Function - This is used to concatenate any string inside any SQL command.
- SQL Numeric Functions - Complete list of SQL functions required to manipulate numbers in SQL.
- SQL String Functions - Complete list of SQL functions required to manipulate strings in SQL.

SQL COUNT Function

SQL **COUNT** function is the simplest function and very useful in counting the number of records, which are expected to be returned by a SELECT statement.

To understand **COUNT** function, consider an **employee_tbl** table, which is having the following records:

```
SQL> 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 count total number of rows in this table, then you can do it as follows:

```
SQL>SELECT COUNT(*) FROM employee_tbl ;
+          +
| COUNT(*) |
+          +
|        7 |
+          +
1 row in set (0.01 sec)
```

Similarly, if you want to count the number of records for Zara, then it can be done as follows:

```
SQL>SELECT COUNT(*) FROM employee_tbl
    -> WHERE name="Zara";
+--------------+
| COUNT(*) |
+--------------+
|        2 |
+--------------+
1 row in set (0.04 sec)
```

NOTE: All the SQL queries are case insensitive, so it does not make any difference if you give ZARA or Zara in WHERE CONDITION.

SQL MAX Function

SQL **MAX** function is used to find out the record with maximum value among a record set.

To understand **MAX** function, consider an **employee_tbl** table, which is having the following records:

```
SQL> 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 maximum value of daily_typing_pages, then you can do so simply using the following command:

```
SQL> SELECT MAX(daily_typing_pages)
    -> FROM employee_tbl;
+----------------------------------+
| MAX(daily_typing_pages) |
+----------------------------------+
|                      350 |
+----------------------------------+
1 row in set (0.00 sec)
```

You can find all the records with maxmimum value for each name using **GROUP BY** clause as follows:

```
SQL> SELECT id, name, MAX(daily_typing_pages)
    -> FROM employee_tbl GROUP BY name;
+--------+--------+-------------------------------+
| id     | name | MAX(daily_typing_pages) |
+--------+--------+-------------------------------+
|     3 | Jack |                     170 |
|     4 | Jill |                     220 |
```

```
|    1 | John |                    250 |
|    2 | Ram  |                    220 |
|    5 | Zara |                    350 |
+      +      +                        +
5 rows in set (0.00 sec)
```

You can use **MIN** Function along with **MAX** function to find out minimum value as well. Try out the following example:

```
SQL> SELECT MIN(daily_typing_pages) least, MAX(daily_typing_pages) max
    -> FROM employee_tbl;
+---------+--------+
| least | max  |
+---------+--------+
|   100 |  350 |
+-------+------+ 1 row in set (0.01 sec)
```

SQL MIN Function

SQL **MIN** function is used to find out the record with minimum value among a record set.

To understand **MIN** function, consider an **employee_tbl** table, which is having the following records:

```
SQL> 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 minimum value of daily_typing_pages, then you can do so simply using the following command:

```
SQL> SELECT MIN(daily_typing_pages)
    -> FROM employee_tbl;
+---------------------------------+
| MIN(daily_typing_pages) |
+---------------------------------+
|                     100 |
+---------------------------------+
1 row in set (0.00 sec)
```

You can find all the records with minimum value for each name using **GROUP BY** clause as follows:

```
SQL> SELECT id, name, work_date, MIN(daily_typing_pages)
    -> FROM employee_tbl GROUP BY name;
+--------+--------+---------------------------------+
| id   | name | MIN(daily_typing_pages) |
+--------+--------+---------------------------------+
|    3 | Jack |                     100 |
```

```
|    4 | Jill |                     220 |
|    1 | John |                     250 |
|    2 | Ram  |                     220 |
|    5 | Zara |                     300 |
+      +      +                         +
5 rows in set (0.00 sec)
```

You can use **MIN** Function along with **MAX** function to find out minimum value as well. Try out the following example:

```
SQL> SELECT MIN(daily_typing_pages) least,
    -> MAX(daily_typing_pages) max
    -> FROM employee_tbl;
+---------+--------+
| least | max   |
+---------+--------+
|   100 |  350 |
+-------+------+ 1 row in set (0.01 sec)
```

SQL AVG Function

SQL **AVG** function is used to find out the average of a field in various records.

To understand **AVG** function, consider an **employee_tbl** table, which is having the following records:

```
SQL> 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 calculate average of all the dialy_typing_pages, then you can do so by using the following command:

```
SQL> SELECT AVG(daily_typing_pages)
    -> FROM employee_tbl;
+---------------------------------+
| AVG(daily_typing_pages) |
+---------------------------------+
|                230.0000 |
+---------------------------------+
1 row in set (0.03 sec)
```

You can take average of various records set using **GROUP BY** clause. Following example will take average all the records related to a single person and you will have average typed pages by every person.

```
SQL> SELECT name, AVG(daily_typing_pages)
    -> FROM employee_tbl GROUP BY name;
+--------+---------------------------------+
```

```
| name | AVG(daily_typing_pages) |
+--------+---------------------------------+
| Jack |                135.0000 |
| Jill |                220.0000 |
| John |                250.0000 |
| Ram  |                220.0000 |
| Zara |                325.0000 |
+--------+---------------------------------+
5 rows in set (0.20 sec)
```

SQL SUM Function

SQL **SUM** function is used to find out the sum of a field in various records.

To understand **SUM** function, consider an **employee_tbl** table, which is having the following records:

```
SQL> 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 calculate total of all the dialy_typing_pages, then you can do so by using the following command:

```
SQL> SELECT SUM(daily_typing_pages)
    -> FROM employee_tbl;
+---------------------------------+
| SUM(daily_typing_pages) |
+---------------------------------+
|                    1610 |
+---------------------------------+
1 row in set (0.00 sec)
```

You can take sum of various records set using **GROUP BY** clause. Following example will sum up all the records related to a single person and you will have total typed pages by every person.

```
SQL> SELECT name, SUM(daily_typing_pages)
    -> FROM employee_tbl GROUP BY name;
+--------+----------------------------------+
| name | SUM(daily_typing_pages) |
+--------+----------------------------------+
| Jack |                      270 |
| Jill |                      220 |
| John |                      250 |
| Ram  |                      220 |
| Zara |                      650 |
+--------+----------------------------------+
5 rows in set (0.17 sec)
```

SQL SQRT Function

SQL **SQRT** function is used to find out the square root of any number. You can Use SELECT statement to find out squre root of any number as follows:

```
SQL>  select SQRT(16);
+-------------+
| SQRT(16) |
+-------------+
| 4.000000 |
+-------------+
1 row in set (0.00 sec)
```

You are seeing float value here because internally SQL will manipulate square root in float data type.

You can use SQRT function to find out square root of various records as well. To understand **SQRT**function in more detail, consider an **employee_tbl** table, which is having the following records:

```
SQL> 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 calculate square root of all the dialy_typing_pages, then you can do so by using the following command:

```
SQL> SELECT name, SQRT(daily_typing_pages)
    -> FROM employee_tbl;
+--------+-----------------------------------+
| name | SQRT(daily_typing_pages) |
+--------+-----------------------------------+
| John |                15.811388 |
| Ram  |                14.832397 |
| Jack |                13.038405 |
```

```
| Jack |                10.000000 |
| Jill |                14.832397 |
| Zara |                17.320508 |
| Zara |                18.708287 |
+--------+----------------------------------+
7 rows in set (0.00 sec)
```

SQL RAND Function

SQL has a **RAND** function that can be invoked to produce random numbers between 0 and 1:

```
SQL>  SELECT RAND( ), RAND( ), RAND( );
+----------------------+----------------------+----------------------+
| RAND( )              | RAND( )              | RAND( )              |
+----------------------+----------------------+----------------------+
| 0.45464584925645 | 0.1824410643265 | 0.54826780459682 |
+----------------------+----------------------+----------------------+
1 row in set (0.00 sec)
```

When invoked with an integer argument, RAND( ) uses that value to seed the random number generator. Each time you seed the generator with a given value, RAND( ) will produce a repeatable series of numbers:

```
SQL>  SELECT RAND(1), RAND( ), RAND( );
+----------------------+----------------------+----------------------+
| RAND(1 )             | RAND( )              | RAND( )              |
+----------------------+----------------------+----------------------+
| 0.18109050223705 | 0.75023211143001 | 0.20788908117254 |
+----------------------+----------------------+----------------------+
1 row in set (0.00 sec)
```

You can use **ORDER BY RAND()** to randomize a set of rows or values as follows:

To understand **ORDER BY RAND()** function, consider an **employee_tbl** table, which is having the following records:

```
SQL> 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, use the following commands:

```
SQL> SELECT * FROM employee_tbl ORDER BY RAND();
+--------+--------+----------------+--------------------------+
| id   | name | work_date  | daily_typing_pages |
+--------+--------+----------------+--------------------------+
|    5 | Zara | 2007-06-06 |                300 |
```

```
|    3 | Jack | 2007-04-06 |                100 |
|    3 | Jack | 2007-05-06 |                170 |
|    2 | Ram  | 2007-05-27 |                220 |
|    4 | Jill | 2007-04-06 |                220 |
|    5 | Zara | 2007-02-06 |                350 |
|    1 | John | 2007-01-24 |                250 |
+------+------+------------+--------------------+
7 rows in set (0.01 sec)

SQL> SELECT * FROM employee_tbl ORDER BY RAND();
+------+------+------------+--------------------+
| id   | name | work_date  | daily_typing_pages |
+------+------+------------+--------------------+
|    5 | Zara | 2007-02-06 |                350 |
|    2 | Ram  | 2007-05-27 |                220 |
|    3 | Jack | 2007-04-06 |                100 |
|    1 | John | 2007-01-24 |                250 |
|    4 | Jill | 2007-04-06 |                220 |
|    3 | Jack | 2007-05-06 |                170 |
|    5 | Zara | 2007-06-06 |                300 |
+------+------+------------+--------------------+
7 rows in set (0.00 sec)
```

SQL CONCAT Function

SQL **CONCAT** function is used to concatenate two strings to form a single string. Try out the following example:

```
SQL> SELECT CONCAT('FIRST ', 'SECOND');
+-----------------------------------+
| CONCAT('FIRST ', 'SECOND') |
+-----------------------------------+
| FIRST SECOND                      |
+-----------------------------------+
1 row in set (0.00 sec)
```

To understand **CONCAT** function in more detail, consider an **employee_tbl** table, which is having the following records:

```
SQL> 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 concatenate all the names employee ID and work_date, then you can do it using the following command:

```
SQL> SELECT CONCAT(id, name, work_date)
    -> FROM employee_tbl;
+--------------------------------------+
| CONCAT(id, name, work_date) |
+--------------------------------------+
| 1John2007-01-24              |
| 2Ram2007-05-27               |
| 3Jack2007-05-06              |
| 3Jack2007-04-06              |
| 4Jill2007-04-06              |
| 5Zara2007-06-06              |
| 5Zara2007-02-06              |
+--------------------------------------+
7 rows in set (0.00 sec)
```

SQL Numeric Function

SQL numeric functions are used primarily for numeric manipulation and/or mathematical calculations. The following table details the numeric functions:

Name	Description
ABS()	Returns the absolute value of numeric expression.
ACOS()	Returns the arccosine of numeric expression. Returns NULL if the value is not in the range -1 to 1.
ASIN()	Returns the arcsine of numeric expression. Returns NULL if value is not in the range -1 to 1
ATAN()	Returns the arctangent of numeric expression.
ATAN2()	Returns the arctangent of the two variables passed to it.
BIT_AND()	Returns the bitwise AND all the bits in expression.
BIT_COUNT()	Returns the string representation of the binary value passed to it.
BIT_OR()	Returns the bitwise OR of all the bits in the passed expression.
CEIL()	Returns the smallest integer value that is not less than passed numeric expression
CEILING()	Returns the smallest integer value that is not less than passed numeric expression
CONV()	Convert numeric expression from one base to another.
COS()	Returns the cosine of passed numeric expression. The numeric expression should be expressed in radians.
COT()	Returns the cotangent of passed numeric expression.

DEGREES()	Returns numeric expression converted from radians to degrees.
EXP()	Returns the base of the natural logarithm (e) raised to the power of passed numeric expression.
FLOOR()	Returns the largest integer value that is not greater than passed numeric expression.
FORMAT()	Returns a numeric expression rounded to a number of decimal places.
GREATEST()	Returns the largest value of the input expressions.
INTERVAL()	Takes multiple expressions exp1, exp2 and exp3 so on.. and returns 0 if exp1 is less than exp2, returns 1 if exp1 is less than exp3 and so on.
LEAST()	Returns the minimum-valued input when given two or more.
LOG()	Returns the natural logarithm of the passed numeric expression.
LOG10()	Returns the base-10 logarithm of the passed numeric expression.
MOD()	Returns the remainder of one expression by diving by another expression.
OCT()	Returns the string representation of the octal value of the passed numeric expression. Returns NULL if passed value is NULL.
PI()	Returns the value of pi
POW()	Returns the value of one expression raised to the power of another expression
POWER()	Returns the value of one expression raised to the power of another expression
RADIANS()	Returns the value of passed expression converted from degrees to radians.
ROUND()	Returns numeric expression rounded to an integer. Can be used to round an expression to a number of decimal points
SIN()	Returns the sine of numeric expression given in radians.
SQRT()	Returns the non-negative square root of numeric expression.
STD()	Returns the standard deviation of the numeric expression.
STDDEV()	Returns the standard deviation of the numeric expression.
TAN()	Returns the tangent of numeric expression expressed in radians.
TRUNCATE()	Returns numeric exp1 truncated to exp2 decimal places. If exp2 is 0, then the result will have no decimal point.

ABS(X)

The ABS() function returns the absolute value of X. Consider the following example:

```
SQL> SELECT ABS(2);
+-----------------------------------------------------------------------------+
| ABS(2)                                                                      |
+-----------------------------------------------------------------------------+
| 2                                                                           |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)

SQL> SELECT ABS(-2);
+-----------------------------------------------------------------------------+
| ABS(2)                                                                      |
+-----------------------------------------------------------------------------+
| 2                                                                           |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

ACOS(X)

This function returns the arccosine of X. The value of X must range between -1 and 1 or NULL will be returned. Consider the following example:

```
SQL> SELECT ACOS(1);
+-----------------------------------------------------------------------------+
| ACOS(1)                                                                     |
+-----------------------------------------------------------------------------+
| 0.000000                                                                    |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

ASIN(X)

The ASIN() function returns the arcsine of X. The value of X must be in the range of -1 to 1 or NULL is returned.

```
SQL> SELECT ASIN(1);
+-----------------------------------------------------------------------------+
| ASIN(1)                                                                     |
+-----------------------------------------------------------------------------+
| 1.5707963267949                                                             |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

ATAN(X)

This function returns the arctangent of X.

```
SQL> SELECT ATAN(1);
+-----------------------------------------------------------------------------+
| ATAN(1)                                                                     |
+-----------------------------------------------------------------------------+
| 0.78539816339745                                                            |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

ATAN2(Y,X)

This function returns the arctangent of the two arguments: X and Y. It is similar to the arctangent of Y/X, except that the signs of both are used to find the quadrant of the result.

```
SQL> SELECT ATAN2(3,6);
+-------------------------------------------------------------------------------+
| ATAN2(3,6)                                                                    |
+-------------------------------------------------------------------------------+
| 0.46364760900081                                                              |
+-------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

BIT_AND(expression)

The BIT_AND function returns the bitwise AND of all bits in expression. The basic premise is that if two corresponding bits are the same, then a bitwise AND operation will return 1, while if they are different, a bitwise AND operation will return 0. The function itself returns a 64-bit integer value. If there are no matches, then it will return 18446744073709551615. The following example performs the BIT_AND function on the PRICE column grouped by the MAKER of the car:

```
SQL> SELECT
          MAKER, BIT_AND(PRICE) BITS
          FROM CARS GROUP BY MAKER
+-------------------------------------------------------------------------------+
|MAKER           BITS                                                           |
+-------------------------------------------------------------------------------+
|CHRYSLER        512                                                            |
|FORD            12488                                                          |
|HONDA           2144                                                           |
+-------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

BIT_COUNT(numeric_value)

The BIT_COUNT() function returns the number of bits that are active in numeric_value. The following example demonstrates using the BIT_COUNT() function to return the number of active bits for a range of numbers:

```
SQL> SELECT
          BIT_COUNT(2) AS TWO,
          BIT_COUNT(4) AS FOUR,
          BIT_COUNT(7) AS SEVEN
+-------+--------+---------+
| TWO | FOUR | SEVEN |
+-------+--------+---------+
|   1 |    1 |     3 |
+-------+--------+---------+
1 row in set (0.00 sec)
```

BIT_OR(expression)

The BIT_OR() function returns the bitwise OR of all the bits in expression. The basic premise of the bitwise OR function is that it returns 0 if the corresponding bits match and 1 if they do not. The function returns a 64-bit integer,

and if there are no matching rows, then it returns 0. The following example performs the BIT_OR() function on the PRICE column of the CARS table, grouped by the MAKER:

```
SQL> SELECT
          MAKER, BIT_OR(PRICE) BITS
          FROM CARS GROUP BY MAKER
+-------------------------------------------------------------------------+
|MAKER           BITS                                                     |
+-------------------------------------------------------------------------+
|CHRYSLER        62293                                                    |
|FORD            16127                                                    |
|HONDA           32766                                                    |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

CEIL(X) CEILING(X)

These functions return the smallest integer value that is not smaller than X. Consider the following example:

```
SQL> SELECT CEILING(3.46);
+-------------------------------------------------------------------------+
| CEILING(3.46)                                                           |
+-------------------------------------------------------------------------+
| 4                                                                       |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)

SQL> SELECT CEIL(-6.43);
+-------------------------------------------------------------------------+
| CEIL(-6.43)                                                             |
+-------------------------------------------------------------------------+
| -6                                                                      |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

CONV(N,from_base,to_base)

The purpose of the CONV() function is to convert numbers between different number bases. The function returns a string of the value N converted from from_base to to_base. The minimum base value is 2 and the maximum is 36. If any of the arguments are NULL, then the function returns NULL. Consider the following example, which converts the number 5 from base 16 to base 2:

```
SQL> SELECT CONV(5,16,2);
+-------------------------------------------------------------------------+
| CONV(5,16,2)                                                            |
+-------------------------------------------------------------------------+
| 101                                                                     |
+-------------------------------------------------------------------------+ 1 row in set (0.00 sec)
```

COS(X)

This function returns the cosine of X. The value of X is given in radians.

```
SQL>SELECT COS(90);
+-------------------------------------------------------------------------+
| COS(90)                                                                 |
+-------------------------------------------------------------------------+
| -0.44807361612917                                                       |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

COT(X)

This function returns the cotangent of X. Consider the following example:

```
SQL>SELECT COT(1);
+-------------------------------------------------------------------------+
| COT(1)                                                                  |
+-------------------------------------------------------------------------+
| 0.64209261593433                                                        |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

DEGREES(X)

This function returns the value of X converted from radians to degrees.

```
SQL>SELECT DEGREES(PI());
+-------------------------------------------------------------------------+
| DEGREES(PI())                                                           |
+-------------------------------------------------------------------------+
| 180.000000                                                              |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

EXP(X)

This function returns the value of e (the base of the natural logarithm) raised to the power of X.

```
SQL>SELECT EXP(3);
+-------------------------------------------------------------------------+
| EXP(3)                                                                  |
+-------------------------------------------------------------------------+
| 20.085537                                                               |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

FLOOR(X)

This function returns the largest integer value that is not greater than X.

```
SQL>SELECT FLOOR(7.55);
+-------------------------------------------------------------------------+
| FLOOR(7.55)                                                             |
```

```
+-----------------------------------------------------------------------------+
| 7                                                                           |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

FORMAT(X,D)

The FORMAT() function is used to format the number X in the following format: ###,###,###.## truncated to D decimal places. The following example demonstrates the use and output of the FORMAT() function:

```
SQL>SELECT FORMAT(423423234.65434453,2);
+-----------------------------------------------------------------------------+
| FORMAT(423423234.65434453,2)                                                |
+-----------------------------------------------------------------------------+
| 423,423,234.65                                                              |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

GREATEST(n1,n2,n3,...)

The GREATEST() function returns the greatest value in the set of input parameters (n1, n2, n3, a nd so on). The following example uses the GREATEST() function to return the largest number from a set of numeric values:

```
SQL>SELECT GREATEST(3,5,1,8,33,99,34,55,67,43);
+-----------------------------------------------------------------------------+
| GREATEST(3,5,1,8,33,99,34,55,67,43)                                         |
+-----------------------------------------------------------------------------+
| 99                                                                          |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

INTERVAL(N,N1,N2,N3,. .)

The INTERVAL() function compares the value of N to the value list (N1, N2, N3, and so on). The function returns 0 if N < N1, 1 if N < N2, 2 if N <N3, and so on. It will return -1 if N is NULL. The value list must be in the form N1 < N2 < N3 in order to work properly. The following code is a simple example of how the INTERVAL() function works:

```
SQL>SELECT INTERVAL(6,1,2,3,4,5,6,7,8,9,10);
+-----------------------------------------------------------------------------+
| INTERVAL(6,1,2,3,4,5,6,7,8,9,10)                                            |
+-----------------------------------------------------------------------------+
| 6                                                                           |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

INTERVAL(N,N1,N2,N3,. .)

The INTERVAL() function compares the value of N to the value list (N1, N2, N3, and so on). The function returns 0 if N < N1, 1 if N < N2, 2 if N <N3, and so on. It will return -1 if N is NULL. The value list must be in the form N1 < N2 < N3 in order to work properly. The following code is a simple example of how the INTERVAL() function works:

```
SQL>SELECT INTERVAL(6,1,2,3,4,5,6,7,8,9,10);
+-------------------------------------------------------------------------+
| INTERVAL(6,1,2,3,4,5,6,7,8,9,10)                                        |
+-------------------------------------------------------------------------+
| 6                                                                       |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

Remember that 6 is the zero-based index in the value list of the first value that was greater than N. In our case, 7 was the offending value and is located in the sixth index slot.

LEAST(N1,N2,N3,N4,.)

The LEAST() function is the opposite of the GREATEST() function. Its purpose is to return the least-valued item from the value list (N1, N2, N3, and so on). The following example shows the proper usage and output for the LEAST() function:

```
SQL>SELECT LEAST(3,5,1,8,33,99,34,55,67,43);
+-------------------------------------------------------------------------+
| LEAST(3,5,1,8,33,99,34,55,67,43)                                        |
+-------------------------------------------------------------------------+
| 1                                                                       |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

LOG(X)
LOG(B,
X)

The single argument version of the function will return the natural logarithm of X. If it is called with two arguments, it returns the logarithm of X for an arbitrary base B. Consider the following example:

```
SQL>SELECT LOG(45);
+-------------------------------------------------------------------------+
| LOG(45)                                                                 |
+-------------------------------------------------------------------------+
| 3.806662                                                                |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)

SQL>SELECT LOG(2,65536);
+-------------------------------------------------------------------------+
| LOG(2,65536)                                                            |
+-------------------------------------------------------------------------+
| 16.000000                                                               |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

LOG10(X)

This function returns the base-10 logarithm of X.

```
SQL>SELECT LOG10(100);
+-----------------------------------------------------------------------------+
| LOG10(100)                                                                  |
+-----------------------------------------------------------------------------+
| 2.000000                                                                    |
+-----------------------------------------------------------------------------+ 1 row in set (0.00 sec)
```

MOD(N,M)

This function returns the remainder of N divided by M. Consider the following example:

```
SQL>SELECT MOD(29,3);
+-----------------------------------------------------------------------------+
| MOD(29,3)                                                                   |
+-----------------------------------------------------------------------------+
| 2                                                                           |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

OCT(N)

The OCT() function returns the string representation of the octal number N. This is equivalent to using CONV(N,10,8).

```
SQL>SELECT OCT(12);
+-----------------------------------------------------------------------------+
| OCT(12)                                                                     |
+-----------------------------------------------------------------------------+
| 14                                                                          |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

PI()

This function simply returns the value of pi. SQL internally stores the full double-precision value of pi.

```
SQL>SELECT PI();
+-----------------------------------------------------------------------------+
| PI()                                                                        |
+-----------------------------------------------------------------------------+
| 3.141593                                                                    |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

POW(X,Y) POWER(X,Y)

These two functions return the value of X raised to the power of Y.

```
SQL> SELECT POWER(3,3);
+-----------------------------------------------------------------------------+
| POWER(3,3)                                                                  |
```

```
+-----------------------------------------------------------------------------+
| 27                                                                          |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

RADIANS(X)

This function returns the value of X, converted from degrees to radians.

```
SQL>SELECT RADIANS(90);
+-----------------------------------------------------------------------------+
| RADIANS(90)                                                                 |
+-----------------------------------------------------------------------------+
|1.570796                                                                     |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

ROUND(X) ROUND(X, D)

This function returns X rounded to the nearest integer. If a second argument, D, is supplied, then the function returns X rounded to D decimal places. D must be positive or all digits to the right of the decimal point will be removed. Consider the following example:

```
SQL>SELECT ROUND(5.693893);
+-----------------------------------------------------------------------------+
| ROUND(5.693893)                                                             |
+-----------------------------------------------------------------------------+
| 6                                                                           |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)

SQL>SELECT ROUND(5.693893,2);
+-----------------------------------------------------------------------------+
| ROUND(5.693893,2)                                                           |
+-----------------------------------------------------------------------------+
| 5.69                                                                        |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

SIGN(X)

This function returns the sign of X (negative, zero, or positive) as -1, 0, or 1.

```
SQL>SELECT SIGN(-4.65);
+-----------------------------------------------------------------------------+
| SIGN(-4.65)                                                                 |
+-----------------------------------------------------------------------------+
| -1                                                                          |
+-----------------------------------------------------------------------------+
```

```
1 row in set (0.00 sec)

SQL>SELECT SIGN(0);
+-----------------------------------------------------------------------------+
| SIGN(0)                                                                     |
+-----------------------------------------------------------------------------+
| 0                                                                           |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)

SQL>SELECT SIGN(4.65);
+-----------------------------------------------------------------------------+
| SIGN(4.65)                                                                  |
+-----------------------------------------------------------------------------+
| 1                                                                           |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

SIN(X)

This function returns the sine of X. Consider the following example:

```
SQL>SELECT SIN(90);
+-----------------------------------------------------------------------------+
| SIN(90)                                                                     |
+-----------------------------------------------------------------------------+
| 0.893997                                                                    |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

SQRT(X)

This function returns the non-negative square root of X. Consider the following example:

```
SQL>SELECT SQRT(49);
+-----------------------------------------------------------------------------+
| SQRT(49)                                                                    |
+-----------------------------------------------------------------------------+
| 7                                                                           |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

STD(expression) STDDEV(expression)

The STD() function is used to return the standard deviation of expression. This is equivalent to taking the square root of the VARIANCE() of expression. The following example computes the standard deviation of the PRICE column in our CARS table:

```
SQL>SELECT STD(PRICE) STD_DEVIATION FROM CARS;
```

```
+-----------------------------------------------------------------------+
| STD_DEVIATION                                                         |
+-----------------------------------------------------------------------+
| 7650.2146                                                             |
+-----------------------------------------------------------------------+
1 row in set (0.00 sec)
```

TAN(X)

This function returns the tangent of the argument X, which is expressed in radians.

```
SQL>SELECT TAN(45);
+-----------------------------------------------------------------------+
| TAN(45)                                                               |
+-----------------------------------------------------------------------+
| 1.619775                                                              |
+-----------------------------------------------------------------------+ 1 row in set (0.00 sec)
```

TRUNCATE(X,D)

This function is used to return the value of X truncated to D number of decimal places. If D is 0, then the decimal point is removed. If D is negative, then D number of values in the integer part of the value is truncated. Consider the following example:

```
SQL>SELECT TRUNCATE(7.536432,2);
+-----------------------------------------------------------------------+
| TRUNCATE(7.536432,2)                                                  |
+-----------------------------------------------------------------------+
| 7.53                                                                  |
+-----------------------------------------------------------------------+
1 row in set (0.00 sec)
```

SQL String Function

SQL string functions are used primarily for string manipulation. The following table details the important string functions:

Name	Description
ASCII()	Returns numeric value of left-most character
BIN()	Returns a string representation of the argument
BIT_LENGTH()	Returns length of argument in bits
CHAR_LENGTH()	Returns number of characters in argument
CHAR()	Returns the character for each integer passed
CHARACTER_LENGTH()	A synonym for CHAR_LENGTH()
CONCAT_WS()	Returns concatenate with separator

CONCAT()	Returns concatenated string
CONV()	Converts numbers between different number bases
ELT()	Returns string at index number
EXPORT_SET()	Returns a string such that for every bit set in the value bits, you get an on string and for every unset bit, you get an off string
FIELD()	Returns the index (position) of the first argument in the subsequent arguments
FIND_IN_SET()	Returns the index position of the first argument within the second argument
FORMAT()	Returns a number formatted to specified number of decimal places
HEX()	Returns a string representation of a hex value
INSERT()	Inserts a substring at the specified position up to the specified number of characters
INSTR()	Returns the index of the first occurrence of substring
LCASE()	Synonym for LOWER()
LEFT()	Returns the leftmost number of characters as specified

LENGTH()	Returns the length of a string in bytes
LOAD_FILE()	Loads the named file
LOCATE()	Returns the position of the first occurrence of substring
LOWER()	Returns the argument in lowercase
LPAD()	Returns the string argument, left-padded with the specified string
LTRIM()	Removes leading spaces
MAKE_SET()	Returns a set of comma-separated strings that have the corresponding bit in bits set
MID()	Returns a substring starting from the specified position
OCT()	Returns a string representation of the octal argument
OCTET_LENGTH()	A synonym for LENGTH()
ORD()	If the leftmost character of the argument is a multi-byte character, returns the code for that character

POSITION()	A synonym for LOCATE()
QUOTE()	Escapes the argument for use in an SQL statement
REGEXP	Pattern matching using regular expressions
REPEAT()	Repeat a string the specified number of times
REPLACE()	Replaces occurrences of a specified string
REVERSE()	Reverses the characters in a string
RIGHT()	Returns the specified rightmost number of characters
RPAD()	Appends string the specified number of times
RTRIM()	Removes trailing spaces
SOUNDEX()	Returns a soundex string
SOUNDS LIKE	Compares sounds
SPACE()	Returns a string of the specified number of spaces
STRCMP()	Compares two strings
SUBSTRING_INDEX()	Returns a substring from a string before the specified number of occurrences of the delimiter
SUBSTRING(), SUBSTR()	Returns the substring as specified
TRIM()	Removes leading and trailing spaces
UCASE()	Synonym for UPPER()
UNHEX()	Converts each pair of hexadecimal digits to a character
UPPER()	Converts to uppercase

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.

```
SQL> SELECT ASCII('2');
+----------------------------------------------------------------------------+
| ASCII('2')                                                                 |
+----------------------------------------------------------------------------+
| 50                                                                         |
+----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

```
SQL> SELECT ASCII('dx');
+-------------------------------------------------------------------------+
| ASCII('dx')                                                             |
+-------------------------------------------------------------------------+
| 100                                                                     |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
SQL> SELECT BIN(12);
+-------------------------------------------------------------------------+
| BIN(12)                                                                 |
+-------------------------------------------------------------------------+
| 1100                                                                    |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

BIT_LENGTH(str)

Returns the length of the string str in bits.

```
SQL> SELECT BIT_LENGTH('text');
+-------------------------------------------------------------------------+
| BIT_LENGTH('text')                                                      |
+-------------------------------------------------------------------------+
| 32                                                                      |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
SQL> SELECT CHAR(77,121,83,81,'76');
+-------------------------------------------------------------------------+
| CHAR(77,121,83,81,'76')                                                 |
+-------------------------------------------------------------------------+ | SQL
|
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
SQL> SELECT CHAR_LENGTH("text");
+-----------------------------------------------------------------------------+
| CHAR_LENGTH("text")                                                         |
+-----------------------------------------------------------------------------+
| 4                                                                           |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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. A numeric argument is converted to its equivalent binary string form; if you want to avoid that, you can use an explicit type cast, as in this example:

```
SQL> SELECT CONCAT('My', 'S', 'QL');
+-----------------------------------------------------------------------------+
| CONCAT('My', 'S', 'QL')                                                     |
+-----------------------------------------------------------------------------+ | SQL
|
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

CONCAT_WS(separator,str1,str2,...)

CONCAT_WS() stands for Concatenate With Separator and is a special form of CONCAT(). The first argument is the separator for the rest of the arguments. The separator is added between the strings to be concatenated. The separator can be a string, as can the rest of the arguments. If the separator is NULL, the result is NULL.

```
SQL> SELECT CONCAT_WS(',','First name','Last Name' );
+-----------------------------------------------------------------------------+
| CONCAT_WS(',','First name','Last Name' )                                    |
+-----------------------------------------------------------------------------+
| First name, Last Name                                                       |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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. The argument N is interpreted as an integer, but may be specified as an integer or a string. The minimum base is 2 and the maximum base is 36. If to_base is a negative number, N is regarded as a signed number. Otherwise, N is treated as unsigned. CONV() works with 64-bit precision.

```
SQL> SELECT CONV('a',16,2);
+-----------------------------------------------------------------------------+
| CONV('a',16,2)                                                              |
```

```
+------------------------------------------------------------------------+
| 1010                                                                   |
+------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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().

```
SQL> SELECT ELT(1, 'ej', 'Heja', 'hej', 'foo');
+------------------------------------------------------------------------+
| ELT(1, 'ej', 'Heja', 'hej', 'foo')                                     |
+------------------------------------------------------------------------+
| ej                                                                     |
+------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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 .,.). The number of bits examined is given by number_of_bits (defaults to 64).

```
SQL> SELECT EXPORT_SET(5,'Y','N',',',4);
+------------------------------------------------------------------------+
| EXPORT_SET(5,'Y','N',',',4)                                            |
+------------------------------------------------------------------------+
| Y,N,Y,N                                                                |
+------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
SQL> SELECT FIELD('ej', 'Hej', 'ej', 'Heja', 'hej', 'foo');
+------------------------------------------------------------------------+
| FIELD('ej', 'Hej', 'ej', 'Heja', 'hej', 'foo')                         |
+------------------------------------------------------------------------+
| 2                                                                      |
+------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
SQL> SELECT FIND_IN_SET('b','a,b,c,d');
+------------------------------------------------------------------------+
| SELECT FIND_IN_SET('b','a,b,c,d')                                      |
+------------------------------------------------------------------------+
```

```
| 2                                                                      |
+------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
SQL> SELECT FORMAT(12332.123456, 4);
+------------------------------------------------------------------------+
| FORMAT(12332.123456, 4)                                                |
+------------------------------------------------------------------------+
| 12,332.1235                                                            |
+------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
SQL> SELECT HEX(255);
+------------------------------------------------------------------------+
| HEX(255)                                                               |
+------------------------------------------------------------------------+
| FF                                                                     |
+------------------------------------------------------------------------+
1 row in set (0.00 sec)

SQL> SELECT 0x616263;
+------------------------------------------------------------------------+
| 0x616263                                                               |
+------------------------------------------------------------------------+
| abc                                                                    |
+------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
SQL> SELECT INSERT('Quadratic', 3, 4, 'What');
+------------------------------------------------------------------------+
| INSERT('Quadratic', 3, 4, 'What')                                      |
+------------------------------------------------------------------------+
| QuWhattic                                                              |
+------------------------------------------------------------------------+
```

```
1 row in set (0.00 sec)
```

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.

```
SQL> SELECT INSTR('foobarbar', 'bar');
+--------------------------------------------------------------------------+
| INSTR('foobarbar', 'bar')                                                |
+--------------------------------------------------------------------------+
| 4                                                                        |
+--------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
SQL> SELECT LEFT('foobarbar', 5);
+--------------------------------------------------------------------------+
| LEFT('foobarbar', 5)                                                     |
+--------------------------------------------------------------------------+
| fooba                                                                    |
+--------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

LENGTH(str)

Returns the length of the string str measured in bytes. A multi-byte character counts as multiple bytes. This means that for a string containing five two-byte characters, LENGTH() returns 10, whereas CHAR_LENGTH() returns 5.

```
SQL> SELECT LENGTH('text');
+--------------------------------------------------------------------------+
| LENGTH('text')                                                           |
+--------------------------------------------------------------------------+
| 4                                                                        |
+--------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

LOAD_FILE(file_name)

Reads the file and returns the file contents as a string. To use this function, the file must be located on the server host, you must specify the full pathname to the file, and you must have the FILE privilege. The file must be readable by all and its size less than max_allowed_packet bytes.

If the file does not exist or cannot be read because one of the preceding conditions is not satisfied, the function returns NULL.

As of SQL 5.0.19, the character_set_filesystem system variable controls interpretation of filenames that are given as literal strings.

```
SQL> UPDATE table_test
    -> SET blob_col=LOAD_FILE('/tmp/picture')
        -> WHERE id=1;
.....................................................
```

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.

```
SQL> SELECT LOCATE('bar', 'foobarbar');
+----------------------------------------------------------------------------+
| LOCATE('bar', 'foobarbar')                                                 |
+----------------------------------------------------------------------------+
| 4                                                                          |
+----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

LOWER(str)

Returns the string str with all characters changed to lowercase according to the current character set mapping.

```
SQL> SELECT LOWER('QUADRATICALLY');
+----------------------------------------------------------------------------+
| LOWER('QUADRATICALLY')                                                     |
+----------------------------------------------------------------------------+
| quadratically                                                              |
+----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
SQL> SELECT LPAD('hi',4,'??');
+----------------------------------------------------------------------------+
| LPAD('hi',4,'??')                                                          |
+----------------------------------------------------------------------------+
| ??hi                                                                       |
+----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

LTRIM(str)

Returns the string str with leading space characters removed.

```
SQL> SELECT LTRIM('  barbar');
+-------------------------------------------------------------------------+
| LTRIM('  barbar')                                                       |
+-------------------------------------------------------------------------+
| barbar                                                                  |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
SQL> SELECT MAKE_SET(1,'a','b','c');
+-------------------------------------------------------------------------+
| MAKE_SET(1,'a','b','c')                                                 |
+-------------------------------------------------------------------------+
| a                                                                       |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
SQL> SELECT OCT(12);
+-------------------------------------------------------------------------+
| OCT(12)                                                                 |
+-------------------------------------------------------------------------+
| 14                                                                      |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

OCTET_LENGTH(str)

OCTET_LENGTH() is a synonym for LENGTH().

ORD(str)

If the leftmost character of the string str is a multi-byte character, returns the code for that character, calculated from the numeric values of its constituent bytes using this formula:

```
  (1st byte code)
+ (2nd byte code . 256)
```

```
+ (3rd byte code . 2562) ...
```

If the leftmost character is not a multi-byte character, ORD() returns the same value as the ASCII() function.

```
SQL> SELECT ORD('2');
+-----------------------------------------------------------------------------+
| ORD('2')                                                                    |
+-----------------------------------------------------------------------------+
| 50                                                                          |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
SQL> SELECT QUOTE('Don\'t!');
+-----------------------------------------------------------------------------+
| QUOTE('Don\'t!')                                                            |
+-----------------------------------------------------------------------------+
| 'Don\'t!'                                                                   |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

NOTE: Please check if your installation has any bug with this function then don't use this function.

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.

```
SQL> SELECT 'ABCDEF' REGEXP 'A%C%%';
+-----------------------------------------------------------------------------+
| 'ABCDEF' REGEXP 'A%C%%'                                                     |
+-----------------------------------------------------------------------------+
| 0                                                                           |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

Another example is:

```
SQL> SELECT 'ABCDE' REGEXP '.*';
+-----------------------------------------------------------------------------+
|  'ABCDE' REGEXP '.*'                                                        |
+-----------------------------------------------------------------------------+
| 1                                                                           |
+-----------------------------------------------------------------------------+
```

```
1 row in set (0.00 sec)
```

Let's see one more example:

```
SQL> SELECT 'new*\n*line' REGEXP 'new\\*.\\*line';
+-------------------------------------------------------------------------+
| 'new*\n*line' REGEXP 'new\\*.\\*line'                                   |
+-------------------------------------------------------------------------+
| 1                                                                       |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
SQL> SELECT REPEAT('SQL', 3);
+-------------------------------------------------------------------------+
| REPEAT('SQL', 3)                                                        |
+-------------------------------------------------------------------------+
| SQLSQLSQL                                                               |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
SQL> SELECT REPLACE('www.mysql.com', 'w', 'Ww');
+-------------------------------------------------------------------------+
| REPLACE('www.mysql.com', 'w', 'Ww')                                     |
+-------------------------------------------------------------------------+
| WwWwWw.mysql.com                                                        |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

REVERSE(str)

Returns the string str with the order of the characters reversed.

```
SQL> SELECT REVERSE('abcd');
+-------------------------------------------------------------------------+
| REVERSE('abcd')                                                         |
+-------------------------------------------------------------------------+
| dcba                                                                    |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

RIGHT(str,len)

Returns the rightmost len characters from the string str, or NULL if any argument is NULL.

```
SQL> SELECT RIGHT('foobarbar', 4);
+----------------------------------------------------------------------------+
| RIGHT('foobarbar', 4)                                                      |
+----------------------------------------------------------------------------+
| rbar                                                                       |
+----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
SQL> SELECT RPAD('hi',5,'?');
+----------------------------------------------------------------------------+
| RPAD('hi',5,'?')                                                           |
+----------------------------------------------------------------------------+
| hi???                                                                      |
+----------------------------------------------------------------------------+ 1 row in set (0.00 sec)
```

RTRIM(str)

Returns the string str with trailing space characters removed.

```
SQL> SELECT RTRIM('barbar   ');
+----------------------------------------------------------------------------+
| RTRIM('barbar   ')                                                         |
+----------------------------------------------------------------------------+
| barbar                                                                     |
+----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

SOUNDEX(str)

Returns a soundex string from str. Two strings that sound almost the same should have identical soundex strings. 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. All non-alphabetic characters in str are ignored. All international alphabetic characters outside the A-Z range are treated as vowels.

```
SQL> SELECT SOUNDEX('Hello');
+----------------------------------------------------------------------------+
| SOUNDEX('Hello')                                                           |
+----------------------------------------------------------------------------+
| H400                                                                       |
+----------------------------------------------------------------------------+ 1 row in set (0.00 sec)
```

expr1 SOUNDS LIKE expr2

This is the same as SOUNDEX(expr1) = SOUNDEX(expr2).

SPACE(N)

Returns a string consisting of N space characters.

```
SQL> SELECT SPACE(6);
+------------------------------------------------------------------------------+
| SELECT SPACE(6)                                                              |
+------------------------------------------------------------------------------+
| '      '                                                                     |
+------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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.

```
SQL> SELECT STRCMP('MOHD', 'MOHD');
+------------------------------------------------------------------------------+
| STRCMP('MOHD', 'MOHD')                                                       |
+------------------------------------------------------------------------------+
| 0                                                                            |
+------------------------------------------------------------------------------+ 1 row in set (0.00 sec)
```

Another example is:

```
SQL> SELECT STRCMP('AMOHD', 'MOHD');
+------------------------------------------------------------------------------+
| STRCMP('AMOHD', 'MOHD')                                                      |
+------------------------------------------------------------------------------+
| -1                                                                           |
+------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

Let's see one more example:

```
SQL> SELECT STRCMP('MOHD', 'AMOHD');
+------------------------------------------------------------------------------+
| STRCMP('MOHD', 'AMOHD')                                                      |
+------------------------------------------------------------------------------+
| 1                                                                            |
+------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

SUBSTRING(str,pos) SUBSTRING(str FROM pos) SUBSTRING(str,pos,len) SUBSTRING(str FROM pos FOR len)

The forms without a len argument return a substring from string str starting at position pos. The forms with a len argument return a substring len characters long from string str, starting at position pos. The forms that use FROM are standard SQL syntax. It is also possible to use a negative value for pos. In this case, the beginning of the substring is pos characters from the end of the string, rather than the beginning. A negative value may be used for pos in any of the forms of this function.

```
SQL> SELECT SUBSTRING('Quadratically',5);
+-------------------------------------------------------------------------+
| SSUBSTRING('Quadratically',5)                                           |
+-------------------------------------------------------------------------+
| ratically                                                               |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)

SQL> SELECT SUBSTRING('foobarbar' FROM 4);
+-------------------------------------------------------------------------+
| SUBSTRING('foobarbar' FROM 4)                                           |
+-------------------------------------------------------------------------+
| barbar                                                                  |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)

SQL> SELECT SUBSTRING('Quadratically',5,6);
+-------------------------------------------------------------------------+
| SUBSTRING('Quadratically',5,6)                                          |
+-------------------------------------------------------------------------+
| ratica                                                                  |
+-------------------------------------------------------------------------+ 1 row in set (0.00 sec)
```

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.

```
SQL> SELECT SUBSTRING_INDEX('www.mysql.com', '.', 2);
+-------------------------------------------------------------------------+
| SUBSTRING_INDEX('www.mysql.com', '.', 2)                                |
+-------------------------------------------------------------------------+
| www.mysql                                                               |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

TRIM([{BOTH | LEADING | TRAILING} [remstr] FROM] str)

TRIM([remstr FROM] str)

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.

```
SQL> SELECT TRIM('  bar   ');
+-------------------------------------------------------------------------+
```

```
| TRIM('  bar   ')                                                |
+                                                                 +
| bar                                                             |
 -----------------------------------------------------------------------------------------
1 row in set (0.00 sec)

SQL> SELECT TRIM(LEADING 'x' FROM 'xxxbarxxx');
+                                                                 +
| TRIM(LEADING 'x' FROM 'xxxbarxxx')                              |
+                                                                 +
| barxxx                                                          |
+                                                                 +
1 row in set (0.00 sec)

SQL> SELECT TRIM(BOTH 'x' FROM 'xxxbarxxx');
+                                                                 +
| TRIM(BOTH 'x' FROM 'xxxbarxxx')                                 |
+                                                                 +
| bar                                                             |
+                                                                 +
1 row in set (0.00 sec)

SQL> SELECT TRIM(TRAILING 'xyz' FROM 'barxxyz');
+                                                                 +
| TRIM(TRAILING 'xyz' FROM 'barxxyz')                             |
+                                                                 +
| barx                                                            |
+                                                                 +
1 row in set (0.00 sec)
```

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.

```
SQL> SELECT UNHEX('4D7953514C');
+-------------------------------------------------------------------------------+
| UNHEX('4D7953514C')                                                           |
+-------------------------------------------------------------------------------+ | SQL
|
+-------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

The characters in the argument string must be legal hexadecimal digits: '0' .. '9', 'A' .. 'F', 'a' .. 'f'. If UNHEX() encounters any non-hexadecimal digits in the argument, it returns NULL.

UPPER(str)

Returns the string str with all characters changed to uppercase according to the current character set mapping.

```
SQL> SELECT UPPER('Allah-hus-samad');
+-------------------------------------------------------------------------------+
| UPPER('Allah-hus-samad')                                                      |
+-------------------------------------------------------------------------------+
| ALLAH-HUS-SAMAD                                                               |
+-------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```