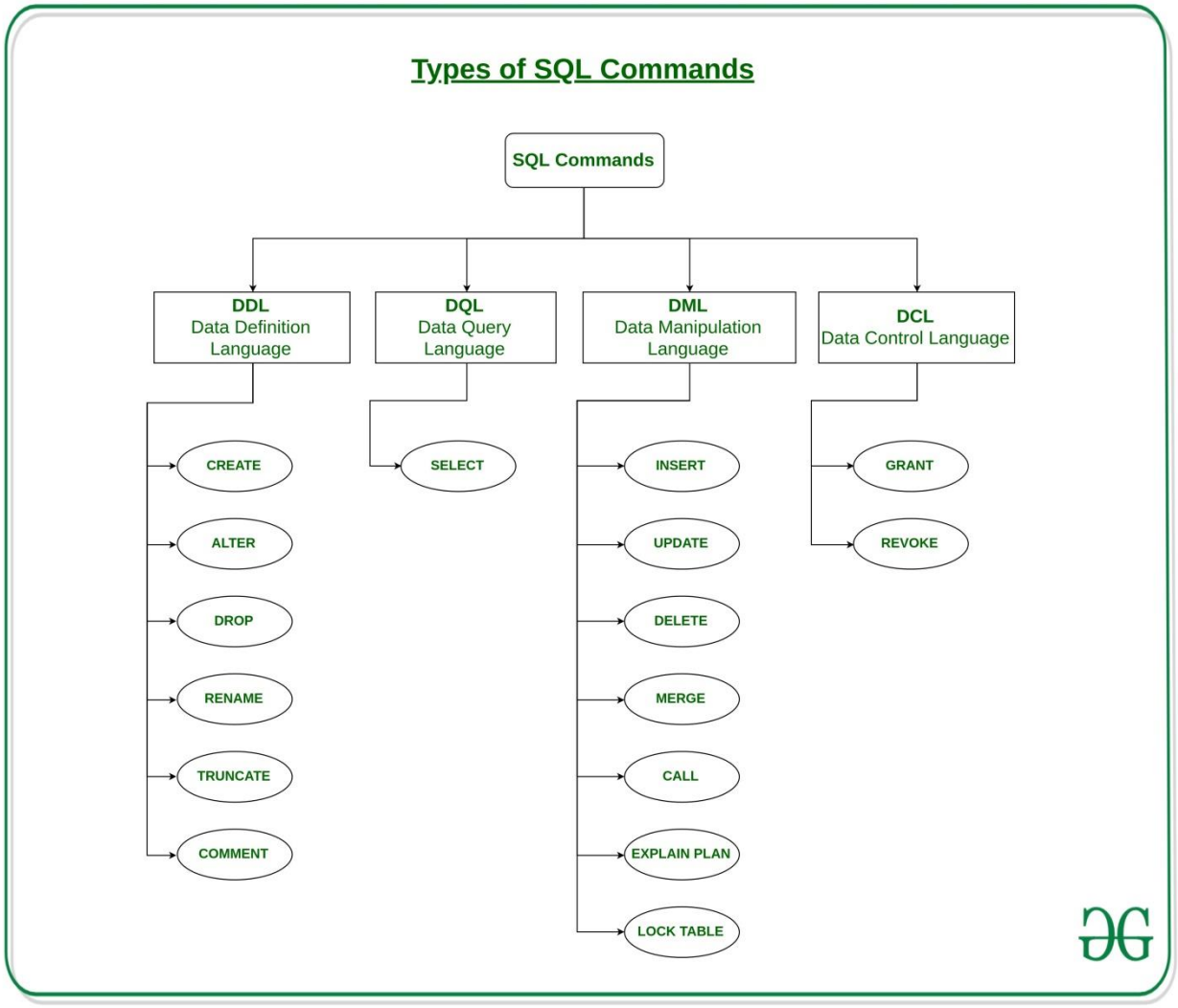
There are four types of SQL Commands

1. Data Definition Language(DDL) – Consists of commands which are used to define the database.
2. Data Manipulation Language(DML) – Consists of commands which are used to manipulate the data present in the database.
3. Data Control Language(DCL) – Consists of commands which deal with the user permissions and controls of the database system.
4. Transaction Control Language(TCL) – Consist of commands which deal with the transaction of the database.



1. **DDL(Data Definition Language) :** DDL or Data Definition Language actually consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database.

Examples of DDL commands:

- **CREATE** – is used to create the database or its objects (like table, index, function, views, store procedure and triggers).
- **DROP** – is used to delete objects from the database.
- **ALTER**-is used to alter the structure of the database.
- **TRUNCATE**–is used to remove all records from a table, including all spaces allocated for the records are removed.
- **COMMENT** –is used to add comments to the data dictionary.
- **RENAME** –is used to rename an object existing in the database.

2. **DQL (Data Query Language) :**
   DML statements are used for performing queries on the data within schema objects. The purpose of DQL Command is to get some schema relation based on the query passed to it.

   Example of DQL:
   - **SELECT** – is used to retrieve data from the a database.
3. **DML(Data Manipulation Language) :** The SQL commands that deals with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements.
   Examples of DML:
   - **INSERT** – is used to insert data into a table.
   - **UPDATE** – is used to update existing data within a table.
   - **DELETE** – is used to delete records from a database table.
4. **DCL(Data Control Language) :** DCL includes commands such as GRANT and REVOKE which mainly deals with the rights, permissions and other controls of the database system.
   Examples of DCL commands:
   - **GRANT**-gives user's access privileges to database.
   - **REVOKE**-withdraw user's access privileges given by using the GRANT command.
5. **TCL(transaction Control Language) :** TCL commands deals with the transaction within the database.
   Examples of TCL commands:
   - **COMMIT**– commits a Transaction.
   - **ROLLBACK**– rollbacks a transaction in case of any error occurs.
   - **SAVEPOINT**–sets a savepoint within a transaction.
   - **SET TRANSACTION**–specify characteristics for the transaction.
6. **DDL:**
   DDL is Data Definition Language which is used to define data structures. For example: create table, alter table are instructions in SQL.
7. **DML:**
   DML is Data Manipulation Language which is used to manipulate data itself. For example: insert, update, delete are instructions in SQL.
8. **Difference between DDL and DML:**

DDL	DML
It stands for Data Definition Language.	It stands for Data Manipulation

	Language.
It is used to create database schema and can be used to define some constraints as well.	It is used to add, retrieve or update the data.
It basically defines the column (Attributes) of the table.	It add or update the row of the table. These rows are called as tuple.
It doesn't have any further classification.	It is further classified into Procedural and Non-Procedural DML.
Basic command present in DDL are CREATE, DROP, RENAME, ALTER etc.	BASIC command present in DML are UPDATE, INSERT, MERGE etc.
DDL does not use WHERE clause in its statement.	While DML uses WHERE clause in its statement.

Difference between DDL and TCL

Last Updated: 24-07-2020

Prerequisite – SQL Commands

1. Data Definition Language (DDL) :
Data Definition Language as the name suggests, it is used to define database schema. For example : create table, alter table are some of the DDL instructions in SQL.

2. Transaction Control Language (TCL) :
Transaction Control Language as the name suggests, contains those commands, which are used to manage transactions within the database.

Difference between DDL and TCL :

S.NO.	DDL	TCL

S.NO.	DDL	TCL
1.	It stands for Data Definition Language.	It stands for Transaction Control Language.
2.	It is used to define data structures or overall database schema.	It contains those commands, which are used to manage transactions within the database.
3.	By using DDL commands, database transactions cannot be handled.	TCL commands are meant to handle database transactions.
4.	Files can be easily maintained by DDL commands.	It manages the different tasks with the important feature, Atomicity.
5.	While writing any query, usually DDL statements are written before TCL statements.	Usually, TCL statements are written before DDL statements.
6.	DDL does not require any log files to maintain the database.	It uses log files to keep track of records of all transactions in a database.

S.NO.	DDL	TCL
7.	Some DDL commands which are frequently used : CREATE, ALTER, DROP.	Some TCL commands which are frequently used : COMMIT, ROLLBACK.

Difference between DML and TCL

Last Updated: 22-06-2020

Prerequisite – DDL, DML, TCL and DCL

1. Data Manipulation Language (DML) :
DML is used to manipulate data in the database. For example, insert, update and delete instructions in SQL.

2. Transaction Control Language (TCL) :
TCL deals with the transactions within the database.

Difference between DML and TCL :

S. NO.	CATEGORY	DML	TCL
1.	Full Form	DML stands for Data Manipulation Language.	TCL stands for Transaction Control Language.
2.	Definition	DML stands for Data Manipulation Language and is used to manipulate data in the database by performing insertion,	Transaction Control Language (TCL) consists of commands that deal with the transactions within databases.

S. NO.	CATEGORY	DML	TCL
		updating and deletion operations.	
3.	Classification	Data Modification Language is further classified into Procedural and Non-Procedural DML.	Transaction Control Language doesn't have any further classifications.
4.	DBMS feature exhibited	It exhibits the feature of easy maintenance (of files).	It exhibits the feature of Atomicity.
5.	Use in Transactions	DML cannot be used for database transactions.	TCL is used for handling database transactions.
6.	Order	DML statements are usually written before TCL statements in a Query.	TCL statements are usually written after DML statements in a Query.

S. NO.	CATEGORY	DML	TCL
7.	Use of Log files	It does not use Log files.	It uses log files to keep a record of all transactions.
8.	Commands	Frequently used commands present in DML are: UPDATE, INSERT, MERGE, SELECT, DELETE, CALL, EXPLAIN PLAN, LOCK TABLE.	Frequently used commands present in TCL are: COMMIT, ROLLBACK, SAVEPOINT, SET TRNSACTION.
9.	Handled by	DML is handled by the Query Compiler and Query Optimizer part of the DBMS architecture.	TCL is handled by the Transaction Manager and Recovery Manager.
10.	Locking	It uses Locks for concurrency control.	It does not use Locks.
11.	WHERE	Most DML statements have WHERE clause to	TCL does not need WHERE

S. NO.	CATEGORY	DML	TCL
	clause	filter them.	clause.
12.	Data Access Paths	DML can be used to explain access paths to data.	TCL cannot explain data access paths.
13.	Call a subprogram	It is used to call PL/SQL or Java subprogram.	It is not used to call subprograms.
14.	Merge operation	We can perform Merge operation using DML.	TCL cannot perform Merge operations.
15.	Trigger	Triggers are fired after DML statements.	TCL is not used for triggers.
16.	Example	Example of SQL query that finds the names of all instructors in the History department : SELECT name FROM instructor	We will use commit command to save the table record permanently. Incase we want to update the name Jolly to sherlock and save it permanently, we would use

S. NO.	CATEGORY	DML	TCL
		WHERE dept_name = ‘History’;	the following, UPDATE STUDENT SET NAME = ‘Sherlock’ WHERE NAME = ‘Jolly’; COMMIT; ROLLBACK;

SQL Commands: Data Definition Language Commands (DDL)

The commands are as follows:

- CREATE
- DROP
- TRUNCATE
- ALTER
- BACKUP DATABASE

CREATE

The ‘CREATE TABLE’ Statement

This statement is used to create a table.

Syntax

CREATE TABLE TableName (Column1 datatype, Column2 datatype, Column3 datatype,

....

ColumnN datatype

);

CREATE TABLE Employee_Info

(

EmployeeID int,

EmployeeName varchar(255),

Emergency ContactName varchar(255),

PhoneNumber int,

Address varchar(255),

City varchar(255),

Country varchar(255)

);

date 14/09/2020

CREATE DATABASE Employee;

show databases;

use Employee;

create table emp(name varchar(25),des varchar(20),dept_no varchar(15),dept_name varchar(25));

describe emp;

insert into emp(name,des,dept_no,dept_name) values ('urmila','professor','cse01','computer science');

alter table emp add salary double(5,2);

describe emp;

insert into emp(name,des,dept_no,dept_name,salary) values ('ankita','asst professor','cse01','computer science',100.2);

insert into emp(name,des,dept_no,dept_name,salary) values ('priya','asst professor','ME01','Mechanical',130.2);

insert into emp(name,des,dept_no,dept_name,salary) values ('Neha','asst professor','CE01','CIVIL',135.2);

alter table emp add dob date;

describe emp;

alter table emp drop dob ;

describe emp;

alter table emp add address varchar(25) not null;

describe emp;

15/09/2020

show databases;

CREATE DATABASE Employee;

use Employee;

create table emp(name varchar(25),des varchar(20),dept_no varchar(15),dept_name varchar(25));

describe emp;

insert into emp(name,des,dept_no,dept_name) values ('urmila','professor','cse01','computer science');

select * from emp;

insert into emp(name,des,dept_no,dept_name) values ('richa','professor','cse01','computer science');

insert into emp(name,des,dept_no,dept_name) values ('abhishek kumar','asst professor','ME01','Mechanical Engg.');

select * from emp;

insert into emp(name,des,dept_no,dept_name) values ('Gurpreet Singh','asst professor','EC01','Electronics Engg.');

insert into emp(name,des,dept_no,dept_name) values ('Amrita Kahira','asst professor','EC01','Electronics Engg.');

insert into emp(name,des,dept_no,dept_name) values ('Md. Wasif','Asst. professor','cse01','computer science');

insert into emp(name,des,dept_no,dept_name) values ('Abhilasha','Asst. professor','cse01','computer science');

insert into emp(name,des,dept_no,dept_name) values ('Mandavi ','professor','cse01','computer science');

select name from emp where dept_no='cse01';

select name,des from emp where dept_no='cse01';

select name,des from emp where des='Asst. Professor';

update emp set name='urmila mahor' where name='urmila';

select * from emp;

update emp set des='Professor' where name='Gurpreet Singh';

select * from emp;

delete from emp where name='Mandavi';

select * from emp order by name desc;

select * from emp order by dept_no ;

select * from emp order by dept_name ;

select count(name) from emp group by dept_name

select count(name) from emp having count(name)>2 order by name;

Q.01Difference between truncate and drop

The 'DROP TABLE' Statement

This statement is used to drop an existing table. When you use this statement, complete information present in the table will be lost.

Syntax

DROP TABLE TableName;

Example

DROP Table Employee_Info;

TRUNCATE

This command is used to delete the information present in the table but does not delete the table. So, once you use this command, your information will be lost, but not the table.

Syntax

TRUNCATE TABLE TableName;

Example
TRUNCATE Table Employee_Info;

The 'ALTER TABLE' Statement

This statement is used to add, delete, modify columns in an existing table.

The 'ALTER TABLE' Statement with ADD/DROP COLUMN

You can use the ALTER TABLE statement with ADD/DROP Column command according to your need. If you wish to add a column, then you will use the ADD command, and if you wish to delete a column, then you will use the DROP COLUMN command.

Syntax
ALTER TABLE TableName
ADD ColumnName Datatype;

ALTER TABLE TableName
DROP COLUMN ColumnName;
Example
--ADD Column BloodGroup:

ALTER TABLE ADD BloodGroup varchar(255);

--DROP Column BloodGroup:

ALTER TABLE Emp DROP COLUMN BloodGroup ;

The 'ALTER TABLE' Statement with ALTER/MODIFY COLUMN
This statement is used to change the datatype of an existing column in a table.

Syntax

ALTER TABLE TableName

MODIFY COLUMN ColumnName Datatype;

Example

--Add a column DOB and change the data type to Date.

ALTER TABLE Emp ADD DOB year;

ALTER TABLE Emp MODIFY DOB date;

BACKUP DATABASE

This statement is used to create a full backup of an existing database.

Syntax
BACKUP DATABASE DatabaseName
TO DISK = 'filepath';
Example
BACKUP DATABASE Employee

TO DISK = 'C:UsersSahitiDesktop';

You can also use a ***differential back up.*** This type of back up only backs up the parts of the database, which have changed since the last complete backup of the database.

Syntax
BACKUP DATABASE DatabaseName
TO DISK = 'filepath'
WITH DIFFERENTIAL;

Example
BACKUP DATABASE Employee

TO DISK = 'C:UsersSahitiDesktop'

WITH DIFFERENTIAL;

SQL Commands: Different Types Of Keys In Database

There are mainly 7 types of Keys, that can be considered in a database. I am going to consider the below tables to explain to you the various keys.

- **Candidate Key –** A set of attributes which can uniquely identify a table can be termed as a Candidate Key. A table can have more than one candidate key, and out of the chosen candidate keys, one key can be chosen as a Primary Key. In the above example, since EmployeeID, InsuranceNumber and PanNumber can uniquely identify every tuple, they would be considered as a Candidate Key.

- **Super Key –** The set of attributes which can uniquely identify a tuple is known as Super Key. So, a candidate key, primary key, and a unique key is a superkey, but vice-versa isn't true.
- **Primary Key –** A set of attributes which are used to uniquely identify every tuple is also a primary key. In the above example, since EmployeeID, InsuranceNumber and PanNumber are candidate keys, any one of them can be chosen as a Primary Key. Here EmployeeID is chosen as the primary key.
- **Alternate Key –** Alternate Keys are the candidate keys, which are not chosen as a Primary key. From the above example, the alternate keys are PanNumber and Insurance Number.
- **Unique Key –** The unique key is similar to the primary key, but allows one NULL value in the column. Here the Insurance Number and the Pan Number can be considered as unique keys.
- **Foreign Key –** An attribute that can only take the values present as the values of some other attribute, is the foreign key to the attribute to which it refers. in the above example, the Employee_ID from the Employee_Information Table is referred to the Employee_ID from the Employee_Salary Table.
- **Composite Key –** A composite key is a combination of two or more columns that identify each tuple uniquely. Here, the Employee_ID and Month-Year_Of_Salary can be grouped together to uniquely identify every tuple in the table.

SQL Commands: Constraints Used In Database

Constraints are used in a database to specify the rules for data in a table. The following are the different types of constraints:

- NOT NULL
- UNIQUE
- CHECK
- DEFAULT
- INDEX

NOT NULL

This constraint ensures that a column cannot have a NULL value.

Example

CREATE TABLE Employee_Info

(

EmployeeID int NOT NULL,

EmployeeName varchar(255) NOT NULL,

Emergency ContactName varchar(255),

PhoneNumber number NOT NULL,

Address varchar(255),

City varchar(255),

Country varchar(255)

);

ALTER TABLE Employee_Info

MODIFY PhoneNumber int NOT NULL;

UNIQUE

This constraint ensures that all the values in a column are unique.

CREATE TABLE Employee_Info

(

EmployeeID int NOT NULL UNIQUE,

EmployeeName varchar(255) NOT NULL,

Emergency ContactName varchar(255),

PhoneNumber int NOT NULL,

Address varchar(255),

City varchar(255),

Country varchar(255)

);

--UNIQUE on Multiple Columns

```
CREATE TABLE Employee_Info
(
EmployeeID int NOT NULL,
EmployeeName varchar(255) NOT NULL,
Emergency ContactName varchar(255),
PhoneNumber int NOT NULL,
Address varchar(255),
City varchar(255),
Country varchar(255),
CONSTRAINT UC_Employee_Info UNIQUE(Employee_ID, PhoneNumber)
);
```

--UNIQUE on ALTER TABLE

```
ALTER TABLE Employee_Info
ADD UNIQUE (Employee_ID);
```

--To drop a UNIQUE constraint

```
ALTER TABLE  Employee_Info
DROP CONSTRAINT UC_Employee_Info;
```

CHECK

This constraint ensures that all the values in a column satisfy a specific condition.

```
CREATE TABLE Employee_Info
(
EmployeeID int NOT NULL,
EmployeeName varchar(255),
Emergency ContactName varchar(255),
PhoneNumber int,
Address varchar(255),
City varchar(255),
Country varchar(255) CHECK (Country=='India')
);

--CHECK Constraint on multiple columns

CREATE TABLE Employee_Info
(
EmployeeID int NOT NULL,
EmployeeName varchar(255),
Emergency ContactName varchar(255),
PhoneNumber int,
Address varchar(255),
City varchar(255),
Country varchar(255) CHECK (Country = 'India' AND Cite = 'Hyderabad')
);
```

--CHECK Constraint on ALTER TABLE

ALTER TABLE Employee_Info

ADD CHECK (Country=='India');

--To give a name to the CHECK Constraint

ALTER TABLE Employee_Info

ADD CONSTRAINT CheckConstraintName CHECK (Country=='India');

--To drop a CHECK Constraint

ALTER TABLE Employee_Info

DROP CONSTRAINT CheckConstraintName;

DEFAULT

This constraint consists of a set of default values for a column when no value is specified.

CREATE TABLE Employee_Info

(

EmployeeID int NOT NULL,

EmployeeName varchar(255),

Emergency ContactName varchar(255),

PhoneNumber int,

Address varchar(255),

City varchar(255),

Country varchar(255) DEFAULT 'India'

);

--DEFAULT Constraint on ALTER TABLE

ALTER TABLE Employee_Info

ADD CONSTRAINT defau_Country

DEFAULT 'India' FOR Country;

--To drop the Default Constraint

ALTER TABLE Employee_Info

ALTER COLUMN Country DROP DEFAULT;

INDEX

This constraint is used to create indexes in the table, through which you can create and retrieve data from the database very quickly.

Syntax

--Create an Index where duplicate values are allowed

CREATE INDEX IndexName

ON TableName (Column1, Column2, ...ColumnN);

--Create an Index where duplicate values are not allowed

CREATE UNIQUE INDEX IndexName

ON TableName (Column1, Column2, ...ColumnN);

Example

CREATE UNIQUE INDEX idex_EmployeeName

ON Persons (EmployeeName);

--To delete an index in a table

DROP INDEX Employee_Info.idex_EmployeeName;

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 discusses a certain topic, you first refer to the index, which lists all the topics alphabetically and are then referred to one or more specific page numbers.

An index helps to speed up **SELECT** queries and **WHERE** clauses, but it slows down data input, with the **UPDATE** and the **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 an ascending or descending order.

Indexes can also be unique, like the **UNIQUE** constraint, in that the index prevents duplicate entries in the column or combination of columns on which there is an index.

The CREATE INDEX Command

CREATE INDEX index_name ON table_name;

Single-Column Indexes

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

CREATE INDEX index_name
ON table_name (column_name);

Unique Indexes

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

```
CREATE UNIQUE INDEX index_name
on table_name (column_name);
```

Composite Indexes

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

```
CREATE INDEX index_name
on table_name (column1, column2);
```

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 the performance may either slow down or improve.

The basic syntax is as follows −

```
DROP INDEX index_name;
```

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 updates or insert operations.
- Indexes should not be used on columns that contain a high number of NULL values.
- Columns that are frequently manipulated should not be indexed.

SQL Commands: Data Manipulation Language Commands (DML)

This section of the article will give you an insight into the commands through which you can manipulate the database. The commands are as follows:

- USE
- INSERT INTO
- UPDATE
- DELETE
- SELECT

Apart from these commands, there are also other manipulative operators/functions such as:

-
- Operators
- Aggregate Functions
- NULL Functions
- Aliases & Case Statement

USE

The USE statement is used to select the database on which you want to perform operations.

Syntax

USE DatabaseName;

Example

USE Employee;

INSERT INTO

This statement is used to insert new records into the table.

Syntax

INSERT INTO TableName (Column1, Column2, Column3, ...,ColumnN)

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

Example

INSERT INTO Employee_Info(EmployeeID, EmployeeName, Emergency ContactName, PhoneNumber, Address, City, Country)

VALUES ('06', 'Sanjana','Jagannath', '9921321141', 'Camel Street House No 12', 'Chennai', 'India');

INSERT INTO Employee_Info

VALUES ('07', 'Sayantini','Praveen', '9934567654', 'Nice Road 21', 'Pune', 'India');

UPDATE

This statement is used to modify the records already present in the table.

Syntax
UPDATE TableName
SET Column1 = Value1, Column2 = Value2, ...
WHERE Condition;
Example
UPDATE Employee_Info

SET EmployeeName = 'Aahana', City= 'Ahmedabad'

WHERE EmployeeID = 1;

DELETE

This statement is used to delete the existing records in a table.

Syntax
DELETE FROM TableName WHERE Condition;
Example
DELETE FROM Employee_Info

WHERE EmployeeName='Preeti';

SELECT

This statement is used to select data from a database and the data returned is stored in a result table, called the **result-set**.

Syntax
SELECT Column1, Column2, ...ColumN
FROM TableName;
SELECT EmployeeID, EmployeeName

FROM Employee_Info;

--(*) is used to select all from the table

SELECT * FROM Employee_Info;

-- To select the number of records to return use:

SELECT TOP 3 * FROM Employee_Info;

Apart from just using the SELECT keyword individually, you can use the following keywords with the SELECT statement:

-
  - DISTINCT
  - ORDER BY
  - GROUP BY
  - HAVING Clause
  - INTO

The 'SELECT DISTINCT' Statement

This statement is used to return only different values.

Syntax

SELECT DISTINCT Column1, Column2, ...ColumnN
FROM TableName;

Example

SELECT DISTINCT PhoneNumber FROM Employee_Info;

The 'ORDER BY' Statement

The 'ORDER BY' statement is used to sort the required results in ascending or descending order. The results are sorted in ascending order by default. Yet, if you wish to get the required results in descending order, you have to use the **DESC** keyword.

Syntax

SELECT Column1, Column2, ...ColumnN
FROM TableName
ORDER BY Column1, Column2, ... ASC|DESC;

Example

-- Select all employees from the 'Employee_Info' table sorted by EmergencyContactName:

SELECT * FROM Employee_Info

```
ORDER BY EmergencyContactName;


-- Select all employees from the 'Employee_Info' table sorted by EmergencyContactName in Descending order:

SELECT * FROM Employee_Info

ORDER BY EmergencyContactName DESC;


-- Select all employees from the 'Employee_Info' table sorted by EmergencyContactName and EmployeeName:

SELECT * FROM Employee_Info

ORDER BY EmergencyContactName, EmployeeName;


/* Select all employees from the 'Employee_Info' table sorted by EmergencyContactName in Descending order and EmployeeName in Ascending order: */

SELECT * FROM Employee_Info

ORDER BY EmergencyContactName ASC, EmployeeName DESC;
```

The 'GROUP BY' Statement

This 'GROUP BY' statement is used with the aggregate functions to group the result-set by one or more columns.

Syntax

```
SELECT Column1, Column2,..., ColumnN
FROM TableName
WHERE Condition
GROUP BY ColumnName(s)
ORDER BY ColumnName(s);
```

Example

```
SELECT COUNT(EmployeeID), City

FROM Employee_Info

GROUP BY City;
```

The 'HAVING' Clause

The 'HAVING' clause is used in SQL because the **WHERE keyword** cannot be used everywhere.

Syntax

```
SELECT ColumnName(s)
FROM TableName
WHERE Condition
GROUP BY ColumnName(s)
HAVING Condition
ORDER BY ColumnName(s);
```

Example

```
SELECT COUNT(EmployeeID), City

FROM Employee_Info

GROUP BY City

HAVING COUNT(EmployeeID) > 2

ORDER BY COUNT(EmployeeID) DESC;
```

The 'SELECT INTO' Statement

The 'SELECT INTO' statement is used to copy data from one table to another.

Syntax

```
SELECT *
INTO NewTable [IN ExternalDB]
FROM OldTable
WHERE Condition;
```

Example

```
-- To create a backup of database 'Employee'

SELECT * INTO EmployeeBackup

FROM Employee;



--To select only few columns from Employee

SELECT EmployeeName, PhoneNumber INTO EmployeeContactDetails

FROM Employee;
```

```
SELECT * INTO BlrEmployee

FROM Employee

WHERE City = 'Bangalore';
```

Operators in SQL

Logical Operators

The Logical operators present in SQL are as follows:

- AND
- OR
- NOT
- BETWEEN
- LIKE
- IN
- EXISTS
- ALL
- ANY

AND Operator

This operator is used to filter records that rely on more than one condition. This operator displays the records, which satisfy all the conditions separated by AND, and give the output TRUE.

Syntax

```
SELECT Column1, Column2, ..., ColumnN

FROM TableName

WHERE Condition1 AND Condition2 AND Condition3 ...;
```

Example

AND Operator

This operator is used to filter records that rely on more than one condition. This operator displays the records, which satisfy all the conditions separated by AND, and give the output TRUE.

Syntax

```
SELECT Column1, Column2, ..., ColumnN
FROM TableName
WHERE Condition1 AND Condition2 AND Condition3 ...;
```

Example

```
SELECT * FROM Employee_Info
WHERE City='Mumbai' AND City='Hyderabad';
```

https://www3.ntu.edu.sg/home/ehchua/programming/sql/MySQL_Beginner.html

```
CREATE TABLE IF NOT EXISTS products (
        productID    INT UNSIGNED  NOT NULL AUTO_INCREMENT,
        productCode  CHAR(3)       NOT NULL DEFAULT '',
        name         VARCHAR(30)   NOT NULL DEFAULT '',
        quantity     INT UNSIGNED  NOT NULL DEFAULT 0,
        price        DECIMAL(7,2)  NOT NULL DEFAULT 99999.99,
        PRIMARY KEY  (productID)
      );
INSERT INTO products VALUES (1001, 'PEN', 'Pen Red', 5000, 1.23);
INSERT INTO products VALUES
        (NULL, 'PEN', 'Pen Blue',  8000, 1.25),
        (NULL, 'PEN', 'Pen Black', 2000, 1.25);
        INSERT INTO products (productCode, name, quantity, price) VALUES
        ('PEC', 'Pencil 2B', 10000, 0.48),
        ('PEC', 'Pencil 2H', 8000, 0.49);
        INSERT INTO products (productCode, name) VALUES ('PEC', 'Pencil HB');
        SELECT * FROM products;
```

productID	productCode	name	quantity	price

```
1001    PEN    Pen Red  5000    1.23
1002    PEN    Pen Blue 8000    1.25
1003    PEN    Pen Black        2000    1.25
1004    PEC    Pencil 2B        10000   0.48
1005    PEC    Pencil 2H        8000    0.49
1006    PEC    Pencil HB        0       99999.99
```

SELECT name, price FROM products WHERE price < 1.0;

```
name      price
Pencil 2B         0.48
Pencil 2H         0.49
```

SELECT name, quantity FROM products **WHERE quantity <= 2000**;

```
name      quantity
Pen Black         2000
Pencil HB         0
```

CAUTION: Do not compare FLOATs (real numbers) for equality ('=' or '<>'), as they are not precise. On the other hand, DECIMAL are precise.

For strings, you could also use '=', '<>', '>', '<', '>=', '<=' to compare two strings (e.g., productCode = 'PEC'). The ordering of string depends on the so-called *collation* chosen. For example,

SELECT name, price FROM products **WHERE productCode = 'PEN'**;

```
name      price
Pen Red  1.23
Pen Blue 1.25
Pen Black         1.25
```

String Pattern Matching - LIKE and NOT LIKE

For strings, in addition to full matching using operators like '=' and '<>', we can perform *pattern matching* using operator LIKE (or NOT LIKE) with wildcard characters. The wildcard '_' matches any single character; '%' matches any number of characters (including zero). For example,

- 'abc%' matches strings beginning with 'abc';
- '%xyz' matches strings ending with 'xyz';
- '%aaa%' matches strings containing 'aaa';
- '___' matches strings containing exactly three characters; and
- 'a_b%' matches strings beginning with 'a', followed by any single character, followed by 'b', followed by zero or more characters.

- SELECT name, price FROM products **WHERE name LIKE 'PENCIL%'**;

```
- name      price
- Pencil 2B  0.48
```

- Pencil 2H 0.49
- Pencil HB 99999.99

- SELECT name, price FROM products **WHERE name LIKE 'P__ %'**;

- name price
- Pen Red 1.23
- Pen Blue 1.25
- Pen Black 1.25

Arithmetic Operators

You can perform arithmetic operations on numeric fields using arithmetic operators, as tabulated below:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
DIV	Integer Division
%	Modulus (Remainder)

Logical Operators - AND, OR, NOT, XOR

You can combine multiple conditions with boolean operators AND, OR, XOR. You can also invert a condition using operator NOT. For examples,

SELECT * FROM products **WHERE quantity >= 5000 AND name LIKE 'Pen %'**;

```
productID       productCode     name    quantity  price
1001    PEN     Pen Red  5000   1.23
1002    PEN     Pen Blue 8000   1.25
```

SELECT * FROM products **WHERE quantity >= 5000 AND price < 1.24 AND name LIKE 'Pen %'**;

```
productID       productCode     name    quantity  price
1001    PEN     Pen Red  5000   1.23
```

SELECT * FROM products **WHERE NOT (quantity >= 5000 AND name LIKE 'Pen %')**;

productID	productCode	name	quantity	price
1003	PEN	Pen Black	2000	1.25
1004	PEC	Pencil 2B	10000	0.48
1005	PEC	Pencil 2H	8000	0.49
1006	PEC	Pencil HB	0	99999.99

IN, NOT IN

You can select from members of a set with IN (or NOT IN) operator. This is easier and clearer than the equivalent AND-OR expression.

```
SELECT * FROM products WHERE name IN ('Pen Red', 'Pen Black');
```

productID	productCode	name	quantity	price
1001	PEN	Pen Red 5000	1.23	
1003	PEN	Pen Black	2000	1.25

```
SELECT * FROM products
    WHERE (price BETWEEN 1.0 AND 2.0) AND (quantity BETWEEN 1000 AND 2000);
```

productID	productCode	name	quantity	price
1003	PEN	Pen Black	2000	1.25

IS NULL, IS NOT NULL

```
SELECT * FROM products WHERE productCode IS NULL;
```

```
SELECT * FROM products WHERE productCode = NULL;
```

ORDER BY Clause

```
SELECT * FROM products WHERE name LIKE 'Pen %' ORDER BY price DESC;
```

productID	productCode	name	quantity	price
1002	PEN	Pen Blue	8000	1.25
1003	PEN	Pen Black	2000	1.25
1001	PEN	Pen Red	5000	1.23

```
SELECT * FROM products WHERE name LIKE 'Pen %' ORDER BY price DESC, quantity;
```

productID	productCode	name	quantity	price
1003	PEN	Pen Black	2000	1.25
1002	PEN	Pen Blue	8000	1.25
1001	PEN	Pen Red	5000	1.23

```
SELECT * FROM products ORDER BY RAND();
```

```
productID   productCode   name        quantity   price
1002        PEN           Pen Blue    8000       1.25
1003        PEN           Pen Black   2000       1.25
1001        PEN           Pen Red     5000       1.23
1006        PEC           Pencil HB   0          99999.99
1004        PEC           Pencil 2B   10000      0.48
1005        PEC           Pencil 2H   8000       0.49
```

LIMIT Clause

A SELECT query on a large database may produce many rows. You could use the LIMIT clause to limit the number of rows displayed, e.g.,

```
SELECT * FROM products ORDER BY price LIMIT 2;
```

```
productID       productCode   name        quantity  price
1004    PEC     Pencil 2B     10000       0.48
1005    PEC     Pencil 2H     8000        0.49
```

```
SELECT * FROM products ORDER BY price LIMIT 2, 1;
```

```
productID       productCode     name        quantity  price
1001    PEN     Pen Red  5000   1.23
```

AS - Alias

You could use the keyword AS to define an *alias* for an identifier (such as column name, table name). The alias will be used in displaying the name. It can also be used as reference. For example,

```
SELECT productID AS ID, productCode AS Code,
       name AS Description, price AS `Unit Price`  -- Define aliases to be used as display names
   FROM products
   ORDER BY ID;
```

```
ID      Code    Description     Unit Price
1001    PEN     Pen Red  1.23
1002    PEN     Pen Blue 1.25
1003    PEN     Pen Black       1.25
1004    PEC     Pencil 2B       0.48
1005    PEC     Pencil 2H       0.49
1006    PEC     Pencil HB       99999.99
```

Function CONCAT()

You can also concatenate a few columns as one (e.g., joining the last name and first name) using function CONCAT(). For example,

```
SELECT CONCAT(productCode, ' - ', name) AS `Product Description`, price FROM products;
```

```
Product Description          price
PEN - Pen Red      1.23
PEN - Pen Blue     1.25
PEN - Pen Black    1.25
PEC - Pencil 2B    0.48
PEC - Pencil 2H    0.49
PEC - Pencil HB    99999.99
```

GROUP BY Clause

The GROUP BY clause allows you to *collapse* multiple records with a common value into groups. For example,

```
SELECT * FROM products ORDER BY productCode, productID;
```

```
productID       productCode     name      quantity  price
1004      PEC     Pencil 2B       10000     0.48
1005      PEC     Pencil 2H       8000      0.49
1006      PEC     Pencil HB       0         99999.99
1001      PEN     Pen Red  5000   1.23
1002      PEN     Pen Blue 8000   1.25
1003      PEN     Pen Black       2000      1.25
```

```
SELECT * FROM products GROUP BY productCode;
```

GROUP BY Aggregate Functions: COUNT, MAX, MIN, AVG, SUM, STD, GROUP_CONCAT

```
SELECT COUNT(*) AS `Count` FROM products;
SELECT productCode, COUNT(*) FROM products GROUP BY productCode;
```

```
productCode     COUNT(*)
PEN             3
PEC             3
```

```
SELECT productCode, COUNT(*) AS count
    FROM products
    GROUP BY productCode
    ORDER BY count DESC;
```

```
productCode     count
PEN     3
PEC     3
```

Besides COUNT(), there are many other GROUP BY aggregate functions such as AVG(), MAX(), MIN() and SUM(). For example,

```
SELECT MAX(price), MIN(price), AVG(price), STD(price), SUM(quantity)
    FROM products;
```

```
MAX(price)      MIN(price)      AVG(price)          STD(price)      SUM(quantity)
99999.990.48    16667.448333    37267.445582443856  33000
```

```
SELECT productCode, MAX(price) AS `Highest Price`, MIN(price) AS `Lowest Price`
    FROM products
    GROUP BY productCode;
```

```
productCode     Highest Price   Lowest Price
PEN     1.25    1.23
PEC     99999.990.48
```

```
SELECT productCode, MAX(price), MIN(price),
        CAST(AVG(price) AS DECIMAL(7,2)) AS `Average`,
        CAST(STD(price) AS DECIMAL(7,2)) AS `Std Dev`,
        SUM(quantity)
    FROM products
    GROUP BY productCode;
```

```
productCode     MAX(price)      MIN(price)      Average Std Dev SUM(quantity)
PEN     1.25    1.23    1.24    0.01    15000
PEC     99999.990.48    33333.6547140.2218000
```

HAVING clause

HAVING is similar to WHERE, but it can operate on the GROUP BY aggregate functions; whereas WHERE operates only on columns.

```
SELECT
        productCode AS `Product Code`,
        COUNT(*) AS `Count`,
        CAST(AVG(price) AS DECIMAL(7,2)) AS `Average`
    FROM products
    GROUP BY productCode
    HAVING Count >=3;
```

```
Product Code    Count   Average
PEN     3       1.24
PEC     3       33333.65
```

WITH ROLLUP

The WITH ROLLUP clause shows the *summary of group summary*, e.g.,

```
SELECT
        productCode,
        MAX(price),
```

```
    MIN(price),
    CAST(AVG(price) AS DECIMAL(7,2)) AS `Average`,
    SUM(quantity)
  FROM products
  GROUP BY productCode
  WITH ROLLUP;
```

```
productCode     MAX(price)      MIN(price)      Average  SUM(quantity)
PEC     99999.990.48    33333.6518000
PEN     1.25    1.23    1.24    15000
NULL    99999.990.48    16667.4533000
```

```
CREATE TABLE IF NOT EXISTS products (
    productID    INT UNSIGNED  NOT NULL AUTO_INCREMENT,
    productCode  CHAR(3)       NOT NULL DEFAULT '',
    name         VARCHAR(30)   NOT NULL DEFAULT '',
    quantity     INT UNSIGNED  NOT NULL DEFAULT 0,
    price        DECIMAL(7,2)  NOT NULL DEFAULT 99999.99,
    PRIMARY KEY  (productID)
  );
INSERT INTO products VALUES (1001, 'PEN', 'Pen Red', 5000, 1.23);
INSERT INTO products VALUES
    (NULL, 'PEN', 'Pen Blue',  8000, 1.25),
    (NULL, 'PEN', 'Pen Black', 2000, 1.25);
    INSERT INTO products (productCode, name, quantity, price) VALUES
    ('PEC', 'Pencil 2B', 10000, 0.48),
    ('PEC', 'Pencil 2H', 8000, 0.49);
    INSERT INTO products (productCode, name) VALUES ('PEC', 'Pencil HB');
    SELECT * FROM products;
    SELECT name, price FROM products WHERE price < 1.0;
```

```
SELECT name, quantity FROM products WHERE quantity <= 2000;

SELECT name, price FROM products WHERE productCode = 'PEN';

SELECT name, price FROM products WHERE name LIKE 'PENCIL%';

SELECT name, price FROM products WHERE name LIKE 'P__ %';

SELECT * FROM products WHERE quantity >= 5000 AND name LIKE 'Pen %';

SELECT * FROM products WHERE quantity >= 5000 AND price < 1.24 AND name
LIKE 'Pen %';

SELECT * FROM products WHERE NOT (quantity >= 5000 AND name LIKE 'Pen %');

SELECT * FROM products WHERE name IN ('Pen Red', 'Pen Black');

SELECT * FROM products

WHERE (price BETWEEN 1.0 AND 2.0) AND (quantity BETWEEN 1000 AND 2000);

SELECT * FROM products WHERE productCode IS NULL;

SELECT * FROM products WHERE productCode = NULL;

SELECT * FROM products WHERE name LIKE 'Pen %' ORDER BY price DESC;
```

productID	productCode	name	quantity	price
1002	PEN	Pen Blue	8000	1.25
1003	PEN	Pen Black	2000	1.25
1001	PEN	Pen Red	5000	1.23

SELECT * FROM products WHERE name LIKE 'Pen %' ORDER BY price DESC, quantity;

productID	productCode	name	quantity	price
1003	PEN	Pen Black	2000	1.25
1002	PEN	Pen Blue	8000	1.25
1001	PEN	Pen Red	5000	1.23

```
SELECT * FROM products ORDER BY RAND();
```

productID	productCode	name	quantity	price

1001	PEN	Pen Red	5000	1.23
1003	PEN	Pen Black	2000	1.25
1006	PEC	Pencil HB	0	99999.99
1002	PEN	Pen Blue	8000	1.25
1004	PEC	Pencil 2B	10000	0.48
1005	PEC	Pencil 2H	8000	0.49

SELECT * FROM products ORDER BY price LIMIT 2;

productID	productCode	name	quantity	price
1004	PEC	Pencil 2B	10000	0.48
1005	PEC	Pencil 2H	8000	0.49

SELECT * FROM products;

productID	productCode	name	quantity	price
1001	PEN	Pen Red	5000	1.23
1002	PEN	Pen Blue	8000	1.25
1003	PEN	Pen Black	2000	1.25
1004	PEC	Pencil 2B	10000	0.48
1005	PEC	Pencil 2H	8000	0.49
1006	PEC	Pencil HB	0	99999.99

SELECT * FROM products ORDER BY price LIMIT 2, 1;

productID	productCode	name	quantity	price
1001	PEN	Pen Red	5000	1.23

SELECT productID AS ID, productCode AS Code,

name AS Description, price AS `Unit Price` -- Define aliases to be used as display names

FROM products

ORDER BY ID;

ID	Code	Description	Unit Price
1001	PEN	Pen Red	1.23
1002	PEN	Pen Blue	1.25
1003	PEN	Pen Black	1.25

1004	PEC	Pencil 2B	0.48
1005	PEC	Pencil 2H	0.49
1006	PEC	Pencil HB	99999.99

SELECT CONCAT(productCode, ' - ', name) AS `Product Description`, price FROM products;

Product Description	price
PEN - Pen Red	1.23
PEN - Pen Blue	1.25
PEN - Pen Black	1.25
PEC - Pencil 2B	0.48
PEC - Pencil 2H	0.49
PEC - Pencil HB	99999.99

SELECT * FROM products ORDER BY productCode, productID;

productID	productCode	name	quantity	price
1004	PEC	Pencil 2B	10000	0.48
1005	PEC	Pencil 2H	8000	0.49
1006	PEC	Pencil HB	0	99999.99
1001	PEN	Pen Red	5000	1.23
1002	PEN	Pen Blue	8000	1.25
1003	PEN	Pen Black	2000	1.25

SELECT COUNT(*) AS `Count` FROM products;

Count
6

SELECT COUNT(*) AS `Total Records` FROM products;

Total Records
6

SELECT productCode, COUNT(*) FROM products GROUP BY productCode;

productCode	COUNT(*)

```
PEN     3
PEC     3
```

```
SELECT productCode, COUNT(*) AS count

FROM products

GROUP BY productCode

ORDER BY count DESC;

SELECT MAX(price), MIN(price), AVG(price), STD(price), SUM(quantity)

FROM products;
```

```
MAX(price)      MIN(price)      AVG(price)              STD(price)              SUM(quantity)
99999.99        0.48            16667.448333333333      37267.445582443856      33000
```

```
SELECT productCode, MAX(price) AS `Highest Price`, MIN(price) AS `Lowest Price`

FROM products

GROUP BY productCode;
```

```
productCode     Highest Price   Lowest Price
PEN             1.25            1.23
PEC             99999.99        0.48
```

```
SELECT productCode, MAX(price), MIN(price),

    CAST(AVG(price) AS DECIMAL(7,2)) AS `Average`,

    CAST(STD(price) AS DECIMAL(7,2)) AS `Std Dev`,

    SUM(quantity)

FROM products

GROUP BY productCode;
```

```
productCode     MAX(price)  MIN(price)  Average     Std Dev     SUM(quan
tity)
```

PEN	1.25	1.23	1.24	0.01	15000
PEC	99999.99	0.48	33333.65	47140.22	18000

```
SELECT    productCode AS `Product Code`,
  COUNT(*) AS `Count`,
  CAST(AVG(price) AS DECIMAL(7,2)) AS `Average`
FROM products
GROUP BY productCode
HAVING Count >=3;
```

Product Code	Count	Average
PEN	3	1.24
PEC	3	33333.65

```
SELECT
  productCode,
  MAX(price),
  MIN(price),
  CAST(AVG(price) AS DECIMAL(7,2)) AS `Average`,
  SUM(quantity)
FROM products
GROUP BY productCode
WITH ROLLUP;
```

productCode	MAX(price)	MIN(price)	Average	SUM(quantity)
PEC	99999.99	0.48	33333.65	18000
PEN	1.25	1.23	1.24	15000
NULL	99999.99	0.48	16667.45	33000

One-To-Many Relationship

```
CREATE TABLE IF NOT EXISTS products (
        productID    INT UNSIGNED  NOT NULL AUTO_INCREMENT,
        productCode  CHAR(3)       NOT NULL DEFAULT '',
        name         VARCHAR(30)   NOT NULL DEFAULT '',
        quantity     INT UNSIGNED  NOT NULL DEFAULT 0,
        price        DECIMAL(7,2)  NOT NULL DEFAULT 99999.99,
        PRIMARY KEY  (productID)
      );
INSERT INTO products VALUES (1001, 'PEN', 'Pen Red', 5000, 1.23);
INSERT INTO products VALUES
        (NULL, 'PEN', 'Pen Blue',  8000, 1.25),
        (NULL, 'PEN', 'Pen Black', 2000, 1.25);
        INSERT INTO products (productCode, name, quantity, price) VALUES
        ('PEC', 'Pencil 2B', 10000, 0.48),
        ('PEC', 'Pencil 2H', 8000, 0.49);
        INSERT INTO products (productCode, name) VALUES ('PEC', 'Pencil HB');
        SELECT * FROM products;
```

productID	productCode	name	quantity	price
1001	PEN	Pen Red	5000	1.23
1002	PEN	Pen Blue	8000	1.25
1003	PEN	Pen Black	2000	1.25
1004	PEC	Pencil 2B	10000	0.48
1005	PEC	Pencil 2H	8000	0.49
1006	PEC	Pencil HB	0	99999.99

```
CREATE TABLE suppliers (
        supplierID  INT UNSIGNED  NOT NULL AUTO_INCREMENT,
```

```
    name        VARCHAR(30)   NOT NULL DEFAULT '',
    phone       CHAR(8)       NOT NULL DEFAULT '',
    PRIMARY KEY (supplierID)
  );
DESCRIBE suppliers;
INSERT INTO suppliers VALUE
    (501, 'ABC Traders', '88881111'),
    (502, 'XYZ Company', '88882222'),
    (503, 'QQ Corp', '88883333');
SELECT * FROM suppliers;
```

supplierID	name	phone
501	ABC Traders	88881111
502	XYZ Company	88882222
503	QQ Corp	88883333

ALTER TABLE

```
ALTER TABLE products   ADD COLUMN supplierID INT UNSIGNED NOT NULL;
DESCRIBE products;
```

productID	productCode	name	quantity	price	supplierID
1001	PEN	Pen Red	5000	1.23	0
1002	PEN	Pen Blue	8000	1.25	0
1003	PEN	Pen Black	2000	1.25	0
1004	PEC	Pencil 2B	10000	0.48	0
1005	PEC	Pencil 2H	8000	0.49	0
1006	PEC	Pencil HB	0	99999.99	0

```
UPDATE products SET supplierID = 501;
```

productID	productCode	name	quantity	price	supplierID
1001	PEN	Pen Red	5000	1.23	501
1002	PEN	Pen Blue	8000	1.25	501

1003	PEN	Pen Black	2000	1.25	501
1004	PEC	Pencil 2B	10000	0.48	501
1005	PEC	Pencil 2H	8000	0.49	501
1006	PEC	Pencil HB	0	99999.99	501

ALTER TABLE products ADD FOREIGN KEY (supplierID) REFERENCES suppliers (supplierID);

DESCRIBE products;

Field	Type	Null	Key	Default	Extra
productID	int unsigned	NO	PRI	NULL	auto_increment
productCode	char(3)	NO			
name	varchar(30)	NO			
quantity	int unsigned	NO		0	
price	decimal(7,2)	NO		99999.99	
supplierID	int unsigned	NO	MUL	NULL	

UPDATE products SET supplierID = 502 WHERE productID = 1004;

SELECT * FROM products;

productID	productCode	name	quantity	price	supplierID
1001	PEN	Pen Red	5000	1.23	501
1002	PEN	Pen Blue	8000	1.25	501
1003	PEN	Pen Black	2000	1.25	501
1004	PEC	Pencil 2B	10000	0.48	502
1005	PEC	Pencil 2H	8000	0.49	501
1006	PEC	Pencil HB	0	99999.99	501

SELECT with JOIN

SELECT command can be used to query and join data from two related tables. For example, to list the product's name (in products table) and supplier's name (in suppliers table), we could join the two table via the two common supplierID columns:

```
SELECT products.name, price, suppliers.name
    FROM products
      JOIN suppliers ON products.supplierID = suppliers.supplierID
```

```
    WHERE price < 0.6;
```

name	price	name
Pencil 2B	0.48	XYZ Company
Pencil 2H	0.49	ABC Traders

```
SELECT products.name, price, suppliers.name
    FROM products, suppliers
    WHERE products.supplierID = suppliers.supplierID
      AND price < 0.6;
```

name	price	name
Pencil 2B	0.48	XYZ Company
Pencil 2H	0.49	ABC Traders

```
SELECT products.name AS `Product Name`, price, suppliers.name AS `Supplier Name`
    FROM products
      JOIN suppliers ON products.supplierID = suppliers.supplierID
    WHERE price < 0.6;
```

Product Name	price	Supplier Name
Pencil 2B	0.48	XYZ Company
Pencil 2H	0.49	ABC Traders

```
SELECT p.name AS `Product Name`, p.price, s.name AS `Supplier Name`
    FROM products AS p
      JOIN suppliers AS s ON p.supplierID = s.supplierID
    WHERE p.price < 0.6;
```

Product Name	price	Supplier Name
Pencil 2B	0.48	XYZ Company
Pencil 2H	0.49	ABC Traders

3.2 Many-To-Many Relationship

```
CREATE TABLE products_suppliers (
        productID   INT UNSIGNED  NOT NULL,
        supplierID  INT UNSIGNED  NOT NULL,
                -- Same data types as the parent tables
        PRIMARY KEY (productID, supplierID),
                -- uniqueness
        FOREIGN KEY (productID)  REFERENCES products  (productID),
        FOREIGN KEY (supplierID) REFERENCES suppliers (supplierID)
      );
DESCRIBE products_suppliers;
INSERT INTO products_suppliers VALUES (1001, 501), (1002, 501),
     (1003, 501), (1004, 502), (1001, 503);
SELECT * FROM products_suppliers;
```

```
productID       supplierID
2001            501
2001            503
2002            501
2003            501
2004            502
```

```
SHOW CREATE TABLE products \G
Create Table: CREATE TABLE `products` (
```

```
  `productID`   int(10) unsigned  NOT NULL AUTO_INCREMENT,
  `productCode` char(3)           NOT NULL DEFAULT '',
  `name`        varchar(30)       NOT NULL DEFAULT '',
  `quantity`    int(10) unsigned  NOT NULL DEFAULT '0',
  `price`       decimal(7,2)      NOT NULL DEFAULT '99999.99',
  `supplierID`  int(10) unsigned   NOT NULL DEFAULT '501',
  PRIMARY KEY (`productID`),
  KEY `supplierID` (`supplierID`),
  CONSTRAINT `products_ibfk_1` FOREIGN KEY (`supplierID`)
    REFERENCES `suppliers` (`supplierID`)
) ENGINE=InnoDB AUTO_INCREMENT=1006 DEFAULT CHARSET=latin1
ALTER TABLE products DROP FOREIGN KEY products_ibfk_1;
SHOW CREATE TABLE products \G
ALTER TABLE products DROP supplierID;
DESC products;
SELECT products.name AS `Product Name`, price, suppliers.name AS `Supplier Name`
      FROM products_suppliers
        JOIN products  ON products_suppliers.productID = products.productID
        JOIN suppliers ON products_suppliers.supplierID = suppliers.supplierID
      WHERE price < 0.6;
SELECT p.name AS `Product Name`, s.name AS `Supplier Name`
      FROM products_suppliers AS ps
        JOIN products AS p ON ps.productID = p.productID
        JOIN suppliers AS s ON ps.supplierID = s.supplierID
      WHERE p.name = 'Pencil 3B';
```

```
SELECT p.name AS `Product Name`, s.name AS `Supplier Name`
    FROM products AS p, products_suppliers AS ps, suppliers AS s
    WHERE p.productID = ps.productID
      AND ps.supplierID = s.supplierID
      AND s.name = 'ABC Traders';
```

3.3 One-to-one Relationship

```
CREATE TABLE product_details (
      productID  INT UNSIGNED   NOT NULL,
            -- same data type as the parent table
      comment    TEXT  NULL,
            -- up to 64KB
      PRIMARY KEY (productID),
      FOREIGN KEY (productID) REFERENCES products (productID)
    );
DESCRIBE product_details;
SHOW CREATE TABLE product_details \G
*************************** 1. row ***************************
    Table: product_details
Create Table: CREATE TABLE `product_details` (
  `productID`  int(10) unsigned  NOT NULL,
  `comment`    text,
  PRIMARY KEY (`productID`),
  CONSTRAINT `product_details_ibfk_1` FOREIGN KEY (`productID`) REFERENCES
`products` (`productID`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

Indexes (or Keys)

Indexes (or Keys) can be created on selected column(s) to facilitate *fast search*. Without index, a "SELECT * FROM products WHERE productID=*x*" needs to match with the productID column of all the records in the products table. If productID column is indexed (e.g., using a binary tree), the matching can be greatly improved (via the binary tree search).
You should index columns which are frequently used in the WHERE clause; and as JOIN columns.
The drawback about indexing is cost and space. Building and maintaining indexes require computations and memory spaces. Indexes facilitate fast search but deplete the performance on modifying the table (INSERT/UPDATE/DELETE), and need to be justified. Nevertheless, relational databases are typically optimized for queries and retrievals, but NOT for updates.
In MySQL, the keyword KEY is synonym to INDEX.

In MySQL, indexes can be built on:

1. a single column (column-index)
2. a set of columns (concatenated-index)
3. on unique-value column (UNIQUE INDEX or UNIQUE KEY)
4. on a prefix of a column for strings (VARCHAR or CHAR), e.g., first 5 characters.

There can be more than one indexes in a table. Index are automatically built on the primary-key column(s).

You can build index via CREATE TABLE, CREATE INDEX or ALTER TABLE.

```
CREATE TABLE employees (
        emp_no      INT UNSIGNED   NOT NULL AUTO_INCREMENT,
        name        VARCHAR(50)    NOT NULL,
        gender      ENUM ('M','F') NOT NULL,
        birth_date  DATE           NOT NULL,
        hire_date   DATE           NOT NULL,
        PRIMARY KEY (emp_no)  -- Index built automatically on primary-key column
    );

DESCRIBE employees;
```

Field	Type	Null	Key	Default	Extra
emp_no	int unsigned	NO	PRI	NULL	auto_increment
name	varchar(50)	NO		NULL	
gender	enum('M','F')	NO		NULL	

```
birth_date          date                NO                  NULL
hire_date date                  NO                  NULL
```

SHOW INDEX FROM employees \G

```
*************************** 1. row ***************************
       Table: employees
  Non_unique: 0
    Key_name: PRIMARY
 Seq_in_index: 1
 Column_name: emp_no
   Collation: A
 Cardinality: 0
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
Index_comment:
     Visible: YES
  Expression: NULL
```

CREATE TABLE departments (

dept_no CHAR(4) NOT NULL,

dept_name VARCHAR(40) NOT NULL,

PRIMARY KEY (dept_no), -- Index built automatically on primary-key column

UNIQUE INDEX (dept_name) -- Build INDEX on this unique-value column

);

DESCRIBE departments;

Field	Type	Null	Key	Default	Extra
dept_no	char(4)	NO	PRI	NULL	
dept_name	varchar(40)	NO	UNI	NULL	

SHOW INDEX FROM departments \G

```
*************************** 2. row ***************************
        Table: departments
   Non_unique: 0
     Key_name: dept_name
 Seq_in_index: 1
  Column_name: dept_name
    Collation: A
  Cardinality: 0
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
Index_comment:
      Visible: YES
   Expression: NULL
```

```
CREATE TABLE dept_emp (
        emp_no     INT UNSIGNED  NOT NULL,
        dept_no    CHAR(4)       NOT NULL,
        from_date  DATE          NOT NULL,
        to_date    DATE          NOT NULL,
        INDEX      (emp_no),          -- Build INDEX on this non-unique-value column
        INDEX      (dept_no),         -- Build INDEX on this non-unique-value column
        FOREIGN KEY (emp_no)  REFERENCES employees (emp_no)
          ON DELETE CASCADE ON UPDATE CASCADE,
        FOREIGN KEY (dept_no) REFERENCES departments (dept_no)
          ON DELETE CASCADE ON UPDATE CASCADE,
        PRIMARY KEY (emp_no, dept_no)  -- Index built automatically
    );
```

DESCRIBE dept_emp;

Field	Type	Null	Key	Default	Extra
emp_no	int unsigned	NO	PRI	NULL	
dept_no	char(4)	NO	PRI	NULL	
from_date	date	NO		NULL	
to_date	date	NO		NULL	

SHOW INDEX FROM dept_emp \G

```
*************************** 4. row ***************************
        Table: dept_emp
   Non_unique: 1
     Key_name: dept_no
  Seq_in_index: 1
  Column_name: dept_no
    Collation: A
  Cardinality: NULL
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
Index_comment:
      Visible: YES
   Expression: NULL
```

Sub-Query

CREATE TABLE suppliers (

supplierID INT UNSIGNED NOT NULL AUTO_INCREMENT,

name VARCHAR(30) NOT NULL DEFAULT '',

phone CHAR(8) NOT NULL DEFAULT '',

PRIMARY KEY (supplierID)

);

INSERT INTO suppliers VALUE

```
        (501, 'ABC Traders', '88881111'),
        (502, 'XYZ Company', '88882222'),
        (503, 'QQ Corp', '88883333');
CREATE TABLE IF NOT EXISTS products (
        productID    INT UNSIGNED  NOT NULL AUTO_INCREMENT,
        productCode  CHAR(3)       NOT NULL DEFAULT '',
        name         VARCHAR(30)   NOT NULL DEFAULT '',
        quantity     INT UNSIGNED  NOT NULL DEFAULT 0,
        price        DECIMAL(7,2)  NOT NULL DEFAULT 99999.99,
        PRIMARY KEY  (productID)
      );
INSERT INTO products VALUES (1001, 'PEN', 'Pen Red', 5000, 1.23);
INSERT INTO products VALUES
        (NULL, 'PEN', 'Pen Blue',  8000, 1.25),
        (NULL, 'PEN', 'Pen Black', 2000, 1.25);
        INSERT INTO products (productCode, name, quantity, price) VALUES
        ('PEC', 'Pencil 2B', 10000, 0.48),
        ('PEC', 'Pencil 2H', 8000, 0.49);
        INSERT INTO products (productCode, name) VALUES ('PEC', 'Pencil HB');
CREATE TABLE products_suppliers (
        productID   INT UNSIGNED  NOT NULL,
        supplierID  INT UNSIGNED  NOT NULL,
                -- Same data types as the parent tables
        PRIMARY KEY (productID, supplierID),
                -- uniqueness
```

```
    FOREIGN KEY (productID)  REFERENCES products  (productID),
    FOREIGN KEY (supplierID) REFERENCES suppliers (supplierID)
  );
INSERT INTO products_suppliers VALUES (1001, 501), (1002, 501),
  (1003, 501), (1004, 502), (1001, 503);
```

Products

productID	productCode	name	quantity	price
1001	PEN	Pen Red	5000	1.23
1002	PEN	Pen Blue	8000	1.25
1003	PEN	Pen Black	2000	1.25
1004	PEC	Pencil 2B	10000	0.48
1005	PEC	Pencil 2H	8000	0.49
1006	PEC	Pencil HB	0	99999.99

suppliers

supplierID	name	phone
501	ABC Traders	88881111
502	XYZ Company	88882222
503	QQ Corp	88883333

products_suppliers

productID	supplierID
1001	501
1001	503
1002	501
1003	501
1004	502

```
SELECT suppliers.name from suppliers
  WHERE suppliers.supplierID
    NOT IN (SELECT DISTINCT supplierID from products_suppliers);


INSERT INTO products_suppliers VALUES (
    (SELECT productID  FROM products  WHERE name = 'Pencil 6B'),
```

```
    (SELECT supplierID FROM suppliers WHERE name = 'QQ Corp'));
DELETE FROM products_suppliers
   WHERE supplierID = (SELECT supplierID FROM suppliers WHERE name = 'QQ Corp');
```

Working with Date and Time

```
CREATE TABLE patients (
    patientID      INT UNSIGNED  NOT NULL AUTO_INCREMENT,
    name           VARCHAR(30)   NOT NULL DEFAULT '',
    dateOfBirth    DATE          NOT NULL,
    lastVisitDate  DATE          NOT NULL,
    nextVisitDate  DATE          NULL,
          -- The 'Date' type contains a date value in 'yyyy-mm-dd'
    PRIMARY KEY (patientID)
  );
INSERT INTO patients VALUES
    (1001, 'Ah Teck', '1991-12-31', '2012-01-20', NULL),
    (NULL, 'Kumar', '2011-10-29', '2012-09-20', NULL),
    (NULL, 'Ali', '2011-01-30', CURDATE(), NULL);
SELECT * FROM patients;
```

patientID	name	dateOfBirth	lastVisitDate	nextVisitDate
1001	Ah Teck	1991-12-31	2012-01-20	NULL
1002	Kumar	2011-10-29	2012-09-20	NULL
1003	Ali	2011-01-30	2020-10-07	NULL

```
SELECT * FROM patients
   WHERE lastVisitDate BETWEEN '2012-09-15' AND CURDATE()
   ORDER BY lastVisitDate;
```

```
patientIDname     dateOfBirth          lastVisitDate        nextVisitDate
1002              Kumar    2011-10-29          2012-09-20           NULL
1003              Ali      2011-01-30          2020-10-07           NULL
```

```
SELECT * FROM patients
    WHERE YEAR(dateOfBirth) = 2011
    ORDER BY MONTH(dateOfBirth), DAY(dateOfBirth);
```

```
patientIDname     dateOfBirth          lastVisitDate        nextVisitDate
1003              Ali      2011-01-30          2020-10-07           NULL
1002              Kumar    2011-10-29          2012-09-20           NULL
```

```
SELECT * FROM patients
    WHERE MONTH(dateOfBirth) = MONTH(CURDATE())
      AND DAY(dateOfBirth) = DAY(CURDATE());
```

```
patientIDname     dateOfBirth          lastVisitDate        nextVisitDate
1003              Ali      2011-01-30          2020-10-07           NULL
1002              Kumar    2011-10-29          2012-09-20           NULL
```

```
SELECT name, dateOfBirth, TIMESTAMPDIFF(YEAR, dateOfBirth, CURDATE()) AS age
    FROM patients
    ORDER BY age, dateOfBirth;
SELECT name, lastVisitDate FROM patients
    WHERE TIMESTAMPDIFF(DAY, lastVisitDate, CURDATE()) > 60;
SELECT name, lastVisitDate FROM patients
    WHERE TO_DAYS(CURDATE()) - TO_DAYS(lastVisitDate) > 60;
SELECT * FROM patients
    WHERE dateOfBirth > DATE_SUB(CURDATE(), INTERVAL 18 YEAR);
```

```
UPDATE patients
    SET nextVisitDate = DATE_ADD(CURDATE(), INTERVAL 6 MONTH)
    WHERE name = 'Ali';
```

Date/Time Functions

MySQL provides these built-in functions for getting the *current* date, time and datetime:

- NOW(): returns the current date and time in the format of 'YYYY-MM-DD HH:MM:SS'.
- CURDATE() (or CURRENT_DATE(), or CURRENT_DATE): returns the current date in the format of 'YYYY-MM-DD'.
- CURTIME() (or CURRENT_TIME(), or CURRENT_TIME): returns the current time in the format of 'HH:MM:SS'.

```
select now(), curdate(), curtime();
```

SQL Date/Time Types

MySQL provides these date/time data types:

- **DATETIME**: stores both date and time in the format of 'YYYY-MM-DD HH:MM:SS'. The valid range is '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. You can set a value using the valid format (e.g., '2011-08-15 00:00:00'). You could also apply functions NOW() or CURDATE() (time will be set to '00:00:00'), but not CURTIME().
- **DATE**: stores date only in the format of 'YYYY-MM-DD'. The range is '1000-01-01' to '9999-12-31'. You could apply CURDATE() or NOW() (the time discarded) on this field.
- **TIME**: stores time only in the format of 'HH:MM:SS'. You could apply CURTIME() or NOW() (the date discarded) for this field.
- **YEAR(4|2)**: in 'YYYY' or 'YY'. The range of years is 1901 to 2155. Use DATE type for year outside this range. You could apply CURDATE() to this field (month and day discarded).
- **TIMESTAMP**: similar to DATETIME but stored the number of seconds since January 1, 1970 UTC (Unix-style). The range is '1970-01-01 00:00:00' to '2037-12-31 23:59:59'. The differences between DATETIME and TIMESTAMP are:
    1. the range,
    2. support for time zone,
    3. TIMESTAMP column could be declared with DEFAULT CURRENT_TIMESTAMP to set the default value to the current date/time. (All other data types' default, including DATETIME, must be a constant and not a function return value). You can also declare a TIMESTAMP column with "ON UPDATE CURRENT_TIMESTAMP" to capture the timestamp of the last update.

The date/time value can be entered manually as a string literal (e.g., '2010-12-31 23:59:59' for DATAETIME). MySQL will issue a warning and insert all zeros (e.g., '0000-00-00 00:00:00' for DATAETIME), if the value of date/time to be inserted is invalid or out-of-range. '0000-00-00' is called a "dummy" date.

More Date/Time Functions

```
SELECT DAYNAME(NOW()), MONTHNAME(NOW()), DAYOFWEEK(NOW()), DAYOFYEAR(NOW());

SELECT DATE_ADD('2012-01-31', INTERVAL 5 DAY);

SELECT DATE_SUB('2012-01-31', INTERVAL 2 MONTH);

CREATE TABLE IF NOT EXISTS `datetime_arena` (

    `description`  VARCHAR(50)  DEFAULT NULL,

    `cDateTime`    DATETIME     DEFAULT '1000-01-01  00:00:00',

    `cDate`        DATE         DEFAULT '1000-01-01 ',

    `cTime`        TIME         DEFAULT '00:00:00',

    `cYear`        YEAR         DEFAULT '0000',

    `cYear2`       YEAR(2)      DEFAULT '0000',

    `cTimeStamp`   TIMESTAMP    DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP

  );

DESCRIBE `datetime_arena`;
```

View

Why views?

- Views can be effective copies of base tables.
- Views can have column names and expressions.
- You can use any clauses in views.
- Views can be used in INSERT/UPDATE/DELETE.
- Views can contain expressions in the select list.
- Views can be views of views.

MySQL Views need Version 5.0 or higher

CREATE TABLE products (prod_id INT NOT NULL AUTO_INCREMENT, prod_name VARCHAR(20) NOT NULL, prod_cost FLOAT NOT NULL DEFAULT 0.0, prod_price FLOAT NOT NULL DEFAULT 0.0, PRIMARY KEY(prod_id));

INSERT INTO products (prod_name, prod_cost, prod_price) VALUES ('Basic Widget',5.95,8.35),('Micro Widget',0.95,1.35),('Mega Widget',99.95,140.00);

CREATE VIEW minimumPriceView AS SELECT prod_name FROM products WHERE prod_cost > 1.00;

SELECT * FROM minimumPriceView;

Transactions

A *atomic transaction* is a set of SQL statements that either ALL succeed or ALL fail. Transaction is important to ensure that there is no *partial* update to the database, given an atomic of SQL statements. Transactions are carried out via COMMIT and ROLLBACK.

```
CREATE TABLE accounts (
        name     VARCHAR(30),
        balance  DECIMAL(10,2)
      );
```

INSERT INTO accounts VALUES ('Paul', 1000), ('Peter', 2000);

SELECT * FROM accounts;

START TRANSACTION;

UPDATE accounts SET balance = balance - 100 WHERE name = 'Paul';

UPDATE accounts SET balance = balance + 100 WHERE name = 'Peter';

COMMIT;

SELECT * FROM accounts;

START TRANSACTION;

UPDATE accounts SET balance = balance - 100 WHERE name = 'Paul';

UPDATE accounts SET balance = balance + 100 WHERE name = 'Peter';

ROLLBACK;

SELECT * FROM accounts;

```
SET autocommit = 0;

UPDATE accounts SET balance = balance - 100 WHERE name = 'Paul';

UPDATE accounts SET balance = balance + 100 WHERE name = 'Peter';

COMMIT;

SELECT * FROM accounts;

UPDATE accounts SET balance = balance - 100 WHERE name = 'Paul';

UPDATE accounts SET balance = balance + 100 WHERE name = 'Peter';

ROLLBACK;

SELECT * FROM accounts;

SET autocommit = 1;   -- Enable autocommit
```

A *transaction* groups a set of operations into a unit that meets the ACID test:

1. Atomicity: If all the operations succeed, changes are *committed* to the database. If any of the operations fails, the entire transaction is *rolled back*, and no change is made to the database. In other words, there is no partial update.
2. Consistency: A transaction transform the database from one consistent state to another consistent state.
3. Isolation: Changes to a transaction are not visible to another transaction until they are committed.
4. Durability: Committed changes are durable and never lost.

More on JOIN

DROP TABLE IF EXISTS t1, t2;

```
CREATE TABLE t1 (
        id      INT PRIMARY KEY,
        `desc`  VARCHAR(30)
      );
CREATE TABLE t2 (
        id      INT PRIMARY KEY,
        `desc`  VARCHAR(30)
```

```
    );
INSERT INTO t1 VALUES
     (1, 'ID 1 in t1'),
     (2, 'ID 2 in t1'),
     (3, 'ID 3 in t1');
INSERT INTO t2 VALUES
     (2, 'ID 2 in t2'),
     (3, 'ID 3 in t2'),
     (4, 'ID 4 in t2');
SELECT * FROM t1;
SELECT * FROM t2;
SELECT *
    FROM t1 INNER JOIN t2;
SELECT *
    FROM t1 INNER JOIN t2 ON t1.id = t2.id;
SELECT *
    FROM t1 INNER JOIN t2 ON t1.id = t2.id;
SELECT *
    FROM t1 JOIN t2 ON t1.id = t2.id;
SELECT *
    FROM t1 CROSS JOIN t2 ON t1.id = t2.id;
SELECT *
    FROM t1 INNER JOIN t2 USING (id);
SELECT *
    FROM t1 INNER JOIN t2 WHERE t1.id = t2.id;
```

```
SELECT *
    FROM t1, t2 WHERE t1.id = t2.id;
```

OUTER JOIN - LEFT JOIN and RIGHT JOIN

```
SELECT *
    FROM t1 LEFT JOIN t2 ON t1.id = t2.id;
SELECT *
    FROM t1 LEFT JOIN t2 USING (id);
SELECT *      FROM t1 RIGHT JOIN t2 ON t1.id = t2.id;
SELECT *
    FROM t1 RIGHT JOIN t2 USING (id);
SELECT t1.id, t1.desc
    FROM t1 LEFT JOIN t2 USING (id)
    WHERE t2.id IS NULL;
SELECT *  FROM t1 LEFT JOIN t2 ON t1.id = t2.id;
SELECT *
    FROM t1 LEFT JOIN t2 USING (id);  -- join-columns have same name
SELECT *
    FROM t1 LEFT JOIN t2 WHERE t1.id = t2.id;
```

Constraints used in MySQL

The following are the most common constraints used in the MySQL:

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

- INDEX
- ENUM
- FOREIGN KEY

NOT NULL Constraint

This constraint specifies that the column cannot have NULL or empty values. The below statement creates a table with NOT NULL constraint.

```
CREATE TABLE Student(Id INTEGER, LastName TEXT NOT NULL, FirstName TEXT NOT NULL, City VARCHAR(35));

INSERT INTO Student VALUES(1, 'Hanks', 'Peter', 'New York');

INSERT INTO Student VALUES(2, NULL, 'Amanda', 'Florida');

ERROR 1048 (23000): Column 'LastName' cannot be null
```

UNIQUE Constraint

This constraint ensures that all values inserted into the column will be unique. It means a column cannot stores duplicate values. MySQL allows us to use more than one column with UNIQUE constraint in a table. The below statement creates a table with a UNIQUE constraint:

```
CREATE TABLE ShirtBrands(Id INTEGER, BrandName VARCHAR(40) UNIQUE, Size VARCHAR(30));

INSERT INTO ShirtBrands(Id, BrandName, Size) VALUES(1, 'Pantaloons', 38), (2, 'Cantabil', 40);

INSERT INTO ShirtBrands(Id, BrandName, Size) VALUES(1, 'Raymond', 38), (2, 'Cantabil', 40);
```

CHECK Constraint

It controls the value in a particular column. It ensures that the inserted value in a column must be satisfied with the given condition. In other words, it determines whether the value associated with the column is valid or not with the given condition.

```
CREATE TABLE Persons (
    ID int NOT NULL,
    Name varchar(45) NOT NULL,
    Age int CHECK (Age>=18)
```

);

```
INSERT INTO Persons(Id, Name, Age)
VALUES (1,'Robert', 28), (2, 'Joseph', 35), (3, 'Peter', 40);

INSERT INTO Persons(Id, Name, Age) VALUES (1,'Robert', 15);
```

DEFAULT Constraint

This constraint is used to set the default value for the particular column where we have not specified any value. It means the column must contain a value, including NULL.

```
CREATE TABLE Persons (  ID int NOT NULL,  Name varchar(45) NOT NULL,  Age int,  City varchar(25) DEFAULT 'India' );
INSERT INTO Persons(Id, Name, Age, City)
VALUES (1,'Robert', 15, 'Florida'),
(2, 'Joseph', 35, 'California'),
(3, 'Peter', 40, 'Alaska');

INSERT INTO Persons(Id, Name, Age) VALUES (1,'Brayan', 15);
```

PRIMARY KEY Constraint

This constraint is used to identify each record in a table uniquely. If the column contains primary key constraints, then it cannot be null or empty. A table may have duplicate columns, but it can contain only one primary key. It always contains unique value into a column.

```
CREATE TABLE Persons (
  ID int NOT NULL PRIMARY KEY,
  Name varchar(45) NOT NULL,
  Age int,
  City varchar(25));

CREATE TABLE Persons (
  ID int  PRIMARY KEY,
  Name varchar(45) NOT NULL,
```

```
Age int,
City varchar(25));


INSERT INTO Persons(Id, Name, Age, City)
VALUES (1,'Robert', 15, 'Florida') ,
(2, 'Joseph', 35, 'California'),
(3, 'Peter', 40, 'Alaska');

INSERT INTO Persons(Id, Name, Age, City)
VALUES (1,'Stephen', 15, 'Florida');
```

AUTO_INCREMENT Constraint

This constraint automatically generates a unique number whenever we insert a new record into the table. Generally, we use this constraint for the primary key field in a table.

```
CREATE TABLE Animals(
id int NOT NULL AUTO_INCREMENT,
name CHAR(30) NOT NULL,
PRIMARY KEY (id));

INSERT INTO Animals (name) VALUES
        ('Tiger'),('Dog'),('Penguin'),
        ('Camel'),('Cat'),('Ostrich');

SELECT * FROM Animals;
```

ENUM Constraint

The ENUM data type in MySQL is a string object. It allows us to limit the value chosen from a list of permitted values in the column specification at the time of table creation. It is short for enumeration, which means that each column may have one of the specified possible values. It uses numeric indexes (1, 2, 3…) to represent string values.

```
CREATE TABLE Shirts (
  id INT PRIMARY KEY AUTO_INCREMENT,
```

```
    name VARCHAR(35),
    size ENUM('small', 'medium', 'large', 'x-large')
);

INSERT INTO Shirts(id, name, size)
VALUES (1,'t-shirt', 'medium'),
(2, 'casual-shirt', 'small'),
(3, 'formal-shirt', 'large');

SELECT * FROM Shirts;
```

INDEX Constraint

This constraint allows us to create and retrieve values from the table very quickly and easily. An index can be created using one or more than one column. It assigns a ROWID for each row in that way they were inserted into the table.

```
CREATE INDEX idx_name ON Shirts(name);

SELECT * FROM Shirts USE INDEX(idx_name);
```

Foreign Key Constraint

This constraint is used to link two tables together. It is also known as the referencing key. A foreign key column matches the primary key field of another table. It means a foreign key field in one table refers to the primary key field of another table.

```
CREATE TABLE Orders (
    Order_ID int NOT NULL PRIMARY KEY,
    Order_Num int NOT NULL,
    Person_ID int,
    FOREIGN KEY (Person_ID) REFERENCES Persons(Person_ID)
);
```

MySQL Create User

The MySQL user is a record in the **USER** table of the MySQL server that contains the login information, account privileges, and the host information for MySQL account. It is essential to create a user in MySQL for accessing and managing the databases.

The MySQL Create User statement allows us to create a new user account in the database server. It provides authentication, SSL/TLS, resource-limit, role, and password management properties for the new accounts. It also enables us to control the accounts that should be initially locked or unlocked.

Syntax

The following syntax is used to create a user in the database server.

```
CREATE USER [IF NOT EXISTS] account_name IDENTIFIED BY 'password';
select user from mysql.user;
create user peter@localhost identified by 'jtp12345';
```

Grant Privileges to the MySQL New User

MySQL server provides multiple types of privileges to a new user account. Some of the most commonly used privileges are given below:

1. **ALL PRIVILEGES:** It permits all privileges to a new user account.
2. **CREATE:** It enables the user account to create databases and tables.
3. **DROP:** It enables the user account to drop databases and tables.
4. **DELETE:** It enables the user account to delete rows from a specific table.
5. **INSERT:** It enables the user account to insert rows into a specific table.
6. **SELECT:** It enables the user account to read a database.
7. **UPDATE:** It enables the user account to update table rows.

```
GRANT ALL PRIVILEGES ON * . * TO peter@localhost;
GRANT CREATE, SELECT, INSERT ON * . * TO peter@localhost;
FLUSH PRIVILEGES;
```

If you want to see the existing privileges for the user, execute the following command.

```
mysql> SHOW GRANTS for username;
DROP USER 'account_name';
DROP USER martin@localhost;
```

MySQL Grant Privilege

MySQL has a feature that provides many control options to the administrators and users on the database. We have already learned how to create a new user using CREATE USER statement in MySQL server. Now, we are going to learn about grant privileges to a user account. MySQL provides GRANT statements to give access rights to a user account.

GRANT Statement

The grant statement enables system administrators to ***assign privileges and roles*** to the MySQL user accounts so that they can use the assigned permission on the database whenever required.

```
CREATE USER john@localhost IDENTIFIED BY 'jtp12345';
SHOW GRANTS FOR john@localhost;
GRANT ALL ON mystudentdb.* TO john@localhost;
```

REVOKE Statement

The revoke statement enables system administrators to ***revoke privileges and roles*** to the MySQL user accounts so that they cannot use the assigned permission on the database in the past.

```
REVOKE ALL, GRANT OPTION FROM john@localhost;
GRANT SELECT, UPDATE, INSERT ON mystudentdb.* TO john@localhost;
SHOW GRANTS FOR john@localhost;
 REVOKE UPDATE, INSERT ON mystudentdb.* FROM john@localhost;

GRANT PROXY ON 'peter@javatpoint' TO 'john'@'localhost' WITH GRANT OPTION;
 SHOW GRANTS FOR 'john'@'localhost';
```

Why we need/use triggers in MySQL?

We need/use triggers in MySQL due to the following features:

- Triggers help us to enforce business rules.
- Triggers help us to validate data even before they are inserted or updated.

- Triggers help us to keep a log of records like maintaining audit trails in tables.
- SQL triggers provide an alternative way to check the integrity of data.
- Triggers provide an alternative way to run the scheduled task.
- Triggers increases the performance of SQL queries because it does not need to compile each time the query is executed.
- Triggers reduce the client-side code that saves time and effort.
- Triggers help us to scale our application across different platforms.
- Triggers are easy to maintain.

Limitations of Using Triggers in MySQL

- MySQL triggers do not allow to use of all validations; they only provide extended validations. **For example**, we can use the NOT NULL, UNIQUE, CHECK and FOREIGN KEY constraints for simple validations.
- Triggers are invoked and executed invisibly from the client application. Therefore, it isn't easy to troubleshoot what happens in the database layer.
- Triggers may increase the overhead of the database server.

Types of Triggers in MySQL?

We can define the maximum six types of actions or events in the form of triggers:

1. **Before Insert:** It is activated before the insertion of data into the table.
2. **After Insert:** It is activated after the insertion of data into the table.
3. **Before Update:** It is activated before the update of data in the table.
4. **After Update:** It is activated after the update of the data in the table.
5. **Before Delete:** It is activated before the data is removed from the table.
6. **After Delete:** It is activated after the deletion of data from the table.

Naming Conventions

Naming conventions are the set of rules that we follow to give appropriate unique names. It saves our time to keep the work organize and understandable. Therefore, **we must use a unique name for each trigger associated with a table**. However, it is a good practice to have the same trigger name defined for different tables.

```
    (BEFOR | AFTER) table_name (INSERT | UPDATE | DELETE)
CREATE TRIGGER trigger_name
   (AFTER | BEFORE) (INSERT | UPDATE | DELETE)
       ON table_name FOR EACH ROW
       BEGIN
      --variable declarations
      --trigger code
      END;

CREATE TABLE employee(
   name varchar(45) NOT NULL,
   occupation varchar(35) NOT NULL,
   working_date date,
   working_hours varchar(10)
);
INSERT INTO employee VALUES
('Robin', 'Scientist', '2020-10-04', 12),
('Warner', 'Engineer', '2020-10-04', 10),
('Peter', 'Actor', '2020-10-04', 13),
('Marco', 'Doctor', '2020-10-04', 14),
('Brayden', 'Teacher', '2020-10-04', 12),
('Antonio', 'Business', '2020-10-04', 11);
Create Trigger before_insert_empworkinghours
BEFORE INSERT ON employee FOR EACH ROW
BEGIN
IF NEW.working_hours < 0 THEN SET NEW.working_hours = 0;
END IF;
END //
INSERT INTO employee VALUES
('Markus', 'Former', '2020-10-08', 14);
INSERT INTO employee VALUES
('Alexander', 'Actor', '2020-10-012', -13);

SHOW TRIGGERS;
```

```
SHOW TRIGGERS IN employeedb;
DROP TRIGGER employeedb.before_update_salaries;
    MySQL BEFORE INSERT TRIGGER
DELIMITER $$
CREATE TRIGGER trigger_name BEFORE INSERT
ON table_name FOR EACH ROW
BEGIN
  variable declarations
  trigger code
END$$
DELIMITER ;

CREATE TABLE employee(
   name varchar(45) NOT NULL,
   occupation varchar(35) NOT NULL,
   working_date date,
   working_hours varchar(10)
);
mysql> DELIMITER //
mysql> Create Trigger before_insert_occupation
BEFORE INSERT ON employee FOR EACH ROW
BEGIN
IF NEW.occupation = 'Scientist' THEN SET NEW.occupation = 'Doctor';
END IF;
END //
INSERT INTO employee VALUES
('Markus', 'Scientist', '2020-10-08', 14);
INSERT INTO employee VALUES
        ('Alexander', 'Actor', '2020-10-012', 13);
    MySQL AFTER INSERT Trigger
DELIMITER $$
CREATE TRIGGER trigger_name AFTER INSERT
ON table_name FOR EACH ROW
BEGIN
```

```
  variable declarations
  trigger code
END$$
DELIMITER ;

CREATE TABLE student_info (
  stud_id int NOT NULL,
  stud_code varchar(15) DEFAULT NULL,
  stud_name varchar(35) DEFAULT NULL,
  subject varchar(25) DEFAULT NULL,
  marks int DEFAULT NULL,
  phone varchar(15) DEFAULT NULL,
  PRIMARY KEY (stud_id)
)
CREATE TABLE student_detail (
  stud_id int NOT NULL,
  stud_code varchar(15) DEFAULT NULL,
  stud_name varchar(35) DEFAULT NULL,
  subject varchar(25) DEFAULT NULL,
  marks int DEFAULT NULL,
  phone varchar(15) DEFAULT NULL,
  Lasinserted Time,
  PRIMARY KEY (stud_id)
);

mysql> DELIMITER //
mysql> Create Trigger after_insert_details
AFTER INSERT ON student_info FOR EACH ROW
BEGIN
INSERT INTO student_detail VALUES (new.stud_id, new.stud_code,
new.stud_name, new.subject, new.marks, new.phone, CURTIME());
END //

INSERT INTO student_info VALUES
```

(10, 110, 'Alexandar', 'Biology', 67, '2347346438');

```
SELECT * FROM student_detail;
```