SQL: Structured query language pronounced as (SEQUEL). This language is used to communicate to database.

Features of SQL:

1. It is a command based language.
2. It is not case sensitive.
3. Every command should end with ';'.
4. Every command starts with "verb".
5. It is similar to English. This language is developed in the year 1972 by "IBM".

Sub language of SQL:

1. DDL (Data Definition Language)
2. DML (Data Manipulation Language)
3. DRL/DQL (Data Retrieval/Query Language)
4. TCL (Transaction Control Language)
5. DCL (Data Control Language)

DDL: This language is used to manage database objects such as table, view, synonym, index and sequence, etc.
CREATE, ALTER, DROP, TRUNCATE, RENAME

DML: This language is used to manipulate the data you have stored
INSERT, UPDATE, DELETE

DRL: This language is used to retrieve the data from the database.
SELECT

TCL: It is used to maintain the transaction of Oracle database.
COMMIT, ROLLBACK, SAVEPOINT

DCL: This language is used to control the access of the data to the users.
GRANT, REVOKE

Table: Table is an object which is used to store some data. In general it is collection of Rows and Columns.

Rules for naming a table:

1. Table name should start with an alphabet, which contains minimum 1 and maximum 30 characters. It should not contain spaces or any special characters such as except _# and 0 to 9.
2. A table can have minimum 1 column, maximum thousand columns.
3. A table can have 0 no. of records and maximum ‘n’ no. of records up to hard disk capacity.
4. Database (MySql or any other) reserved keywords and words should not be used as column names or table names.
5. The rules which we are following for table name, the same rules are applicable for column names.
6. The numeric precision for column must be 1 to 38.

Create command: This command is used to create a table.
CREATE TABLE <TABLE_NAME>(COL_NAME1 DATATYPE(SIZE),
COL_NAME2 DATATYPE(SIZE), COL_NAME3 DATATYPE(SIZE),…
,COL_NAMEn Datatype(size));

Schema.sql

```
/* source filename to run from command line */
/* source [FILENAME]
USE Test;

/* current database */
SELECT DATABASE();

DROP DATABASE IF EXISTS test;
CREATE DATABASE test;
USE test;

SHOW TABLES;

DROP TABLE IF EXISTS dept;
DROP TABLE IF EXISTS salgrade;
DROP TABLE IF EXISTS emp;

CREATE TABLE salgrade(
  grade int(4) primary key,
  losal decimal(10,2),
  hisal decimal(10,2)
);

CREATE TABLE dept(
  deptno int(2) primary key,
  dname varchar(50) not null,
  location varchar(50) not null
);

CREATE TABLE emp(
  empno int(4) primary key,
  ename varchar(50) not null,
  job varchar(50) not null,
  mgr int(4),
  hiredate date,
  sal decimal(10,2),
  comm decimal(10,2),
  deptno int(2)
);

ALTER TABLE emp ADD CONSTRAINT fk_dept FOREIGN KEY (deptno)
REFERENCES dept (deptno);

ALTER TABLE emp ADD CONSTRAINT fk_mgr FOREIGN KEY (mgr)
  REFERENCES emp (empno);
```

Insert Command:

INSERT INTO<TABLE_NAME>VALUES(VAL1,VAL2,VAL3,…………VALn);

INSERT INTO <TABLE_NAME> (COL1,COL2,…..COLn) VALUES (VAL1,VAL2,…….VALn);

```
insert into dept values (10,'Accounting','New York');
insert into dept values (20,'Research','Dallas');
insert into dept values (30,'Sales','Chicago');
insert into dept values (40,'Operations','Boston');
insert into dept values (50,'Finance','Tempe');

insert into emp values
(7369,'SMITH','CLERK',7902,'93/6/13',800,0.00,20);
insert into emp values
(7499,'ALLEN','SALESMAN',7698,'98/8/15',1600,300,10);
insert into emp values
(7521,'ALLEN','SALESMAN',7698,'96/3/26',1250,500,30);
insert into emp values
(7566,'JONES','MANAGER',7839,'95/10/31',2975,null,20);
insert into emp values
(7698,'BLAKE','MANAGER',7839,'92/6/11',2850,null,30);
insert into emp values
(7782,'CLARK','MANAGER',7839,'93/5/14',2450,null,10);
insert into emp values
(7788,'SCOTT','ANALYST',7566,'96/3/5',3000,null,20);
insert into emp values
(7839,'KEVIN','PRESIDENT',null,'90/6/9',5000,0,40);
insert into emp values
(7844,'KEVIN','SALESMAN',7698,'95/6/4',1500,0,30);
insert into emp values
(7876,'KEVIN','CLERK',7788,'99/6/4',1100,null,20);
insert into emp values
(7900,'JAMES','CLERK',7698,'00/6/23',950,null,20);
insert into emp values
(7934,'FORD','CLERK',7782,'00/1/21',1300,null,10);
insert into emp values
(7902,'FORD','ANALYST',7566,'97/12/5',3000,null,20);
insert into emp values
(7654,'MARTIN','SALESMAN',7698,'98/12/5',1250,1400,40);

insert into salgrade values (1,700,2222);
insert into salgrade values (2,1201,3333);
insert into salgrade values (3,1401,4444);
insert into salgrade values (4,2001,5555);
insert into salgrade values (5,3001,6666);
insert into salgrade (losal,hisal,grade) values (4001,8888,
6);
```

Select: This command is used to return the data from the table.

SELECT * FROM <TABLE_NAME>;

```
select * from emp; // * represent ALL
```

Note: where we use * to indicate all the fields information (ALL the columns and the rows are displayed).

Selecting specific columns:

```
select empno, ename, deptno from emp;
```

Distinct Keyword: it is used to display distinct values (unique values). Duplicates are suppressed.

update: The command is used to change / modify the data present in the table.
Update <TABLE_NAME> set <COL_NAME> = <VALUE> where <CONDITION>;

```
update emp set job = 'analyst', deptno = 30 where empno =
7876;

update salgrade set losal = 1000;

update dept set deptno = 25;

update emp set mgr = NULL where empno = 7876;
```

Note: when where clause is not used, all the rows are updated.

Delete: This command is used to remove the complete row from the table.
Delete from <table_name> where <condition>;

```
delete from salgrade
  where grade > 5;

delete from salgrade
  where grade > 3 and losal < 3000;
```

```
delete from salgrade;
```

Note: if "**where**" clause is not used in the delete from command, then all the rows gets deleted.

Logical Operators: There are three logical operators. They are

1. AND
2. OR
3. NOT

```
select * from salgrade where grade < 4 and hisal > 4000;

select * from emp where deptno = 20 or deptno = 30;

select ename, deptno from emp
  where deptno not between 10 and 30;
```

Note: AND operator will return the rows when all the conditions are satisfied.

Between: operator is used to display the rows that are falling in the given range of values.

```
select * from emp
  where deptno between 10 and 30;
```

Note: Extreme values are included. That is 10 and 30 are inclusive.
Always specify first lower limit first and then higher limit.

IN Operator:

1. IN operator will return the rows when the values are matching in the list.
2. IN operator can be used as a replacement of OR operator.

```
select * from emp where deptno in (20, 30);
select * from emp where deptno = 20 or deptno = 30;
```

Select1.sql

```
/* To write sql output to a file do the following before you run the script */
/* tee e:\tanla\mysql\output.txt */
/* Once done you can disable output by using */
/* notee */

use test;

desc emp;
desc dept;
desc salgrade;

select * from emp;

select * from from dept;

select * from salgrade;

select * from emp where deptno = 10;

select * from emp where hiredate = '98/8/15';

select * from emp where deptno =
  (select deptno from dept where dname = "Sales");

/* Aliases can be useful when:

There are more than one table involved in a query
Functions are used in the query
Column names are big or not very readable
Two or more columns are combined together */

select empno, ename as EmployeeName, deptno as 'Department
Number' from emp;

select empno, concat(ename,', ',job,', ',mgr,', ',hiredate)
  as 'Employee details' from emp;

select job from Emp;
select distinct job from Emp;

select * from salgrade
  where grade between 2 and 4;
select * from salgrade
  where grade >= 1;   // Test with >= & <=

select * from emp
  where deptno between 10 and 30;

select ename, deptno from emp
  where deptno not between 10 and 30;

select * from emp
  where deptno in (20, 30);

select * from emp
  where deptno = 20 or deptno = 30;

select * from emp
  where ename between 'K' and 'A';

update emp set ename = 'K'
  where empno = 7876;

select * from emp
  where ename between 'A' and 'K';

update emp set ename = 'Kevin'
  where empno = 7876;

select * from emp
  where ename NOT between 'A' and 'K';
select * from salgrade
  where grade < 3;

select * from salgrade
  where grade < 4 and hisal > 4000;
```

Pattern Matching Operator: They are two pattern matching operator.

1. Percentage (%)
2. Under score (_)

Percentage (%): This command is used to select the characters (more than one).

```
select * from emp  where ename like 'k%n';
```

Under Score: This command is used to select the letter (one underscore for one character).

```
select * from emp  where ename like 'A____'
```

select2.sql

```
select * from emp where ename like 'a%';

select * from emp where ename like '%n';

select * from emp where ename like '%in';

select * from emp where ename like '%l_%';

select * from emp where ename like 'A___n';

select * from emp where ename like 'k%n';

select * from emp where ename not like 'k%n';

/* does not work in MYSQL */
select * from emp where ename like '[km]%n';

select * from emp where ename like 'k%n' or ename like
'm%n';

SELECT * FROM emp WHERE ename REGEXP '[km]';

SELECT * FROM emp WHERE ename rlike '[km]';
SELECT * FROM emp WHERE ename REGEXP '^[km]';

SELECT * FROM emp WHERE ename REGEXP '^[AF]';
SELECT * FROM emp WHERE ename rlike '[AF]'

/* Starting with A and all characters upto starting with K
*/
SELECT * FROM emp WHERE ename REGEXP '^[A-K]';

SELECT * FROM emp WHERE ename REGEXP '[AK]';

select * from emp where empno like '7%8';
```

Subquery

A MySQL subquery is a query that is nested inside another query such as SELECT, INSERT, UPDATE or DELETE. In addition, a MySQL subquery can be nested inside another subquery. A MySQL subquery is also called an inner query while the query that contains the subquery is called an outer query.

subqueries.sql

```
select * from emp where deptno =
  (select deptno from dept where dname = "Sales");

select * from emp where deptno in
  (select deptno from dept where job = "Manager");

select * from emp where deptno not in
  (select deptno from dept where job = "Manager");

/* Test with greater than (>), less than < etc */
select ename, empno, sal
  from emp
where sal = (
  select max(sal) from emp
);
```

```
select avg(sal) from emp;

select ename, empno, sal
  from emp
where sal > (
  select avg(sal) from emp
);

select ename, empno, sal, avg(sal)
  from emp
where sal > (
  select avg(sal) from emp
);

select deptno, dname
  from dept
where
  deptno not in (select distinct(deptno) from emp);

select deptno, dname
  from dept
where
  deptno in (select distinct(deptno) from emp);
```

Note: To see warnings enter

```
Show warnings;
```

DDL (Data Definition Language):

1. Create
2. Alter
3. Drop
4. Truncate
5. Rename

Lets create a customer table first and then use alter statements to change the structure.

```
create table customer(
  custid int(4) NOT NULL,
  aadharno varchar(20) NOT NULL UNIQUE,
  phone varchar(10) NOT NULL
);

insert into customer values(21,'Adhar21',1234567890);
insert into customer values(31,'Adhar31',9876543210);
```

ALTER: By using ALTER command we can perform the following task.
Adding new columns or Constraints
Dropping an existing column or Constraint
Modifying a column
Renaming a column

Adding new Columns:
Syntax: ALTER TABLE <TABLE_NAME> ADD (COL1_NAME DATA TYPE(SIZE),(COL2_NAME DATA TYPE(SIZE));

```
alter table customer add(city varchar(20));
alter table customer add(state varchar(20), country varchar(20));

desc customer;
```

// works in mysql 5.7 and above

```
alter table customer add column city2 varchar(10) after city;
alter table customer add column city3 varchar(10) not null after city2;
```

What do you think you will happen if we execute the following after the above alter statements are executed?

```
insert into customer values(41,'Adhar41',5566778899);
```

Note: New column(s) can be added only at last or in between (depends on the MySql version)

The new column(s) will have null values.

ADDING A CONSTRAINT TO THE TABLE

Syntax: ALTER TABLE <TABLE_NAME>
ADD constraint constraint-name constraint-type (field-names);

```
alter table customer add primary key(custid);
alter table customer add constraint uk_phone UNIQUE(phone);
```

Modifying a column: (increasing/decreasing the size of columns)

Syntax: ALTER TABLE <TABLE_NAME> MODIFY(COL1_NAME DATA TYPE(SIZE));

```
alter table customer modify column city varchar(20);
update customer set city = "Hyderabad city";
desc customer;
alter table customer modify column city varchar(10);
```

Note: We can increase/decrease the size of the column.
We can decrease the column size only when existing column values can fit into new size.
By using modify keyword we can change the data type of a column.
Column should be empty to change its data type.

RENAMING A COLUMN:

Syntax: ALTER TABLE <TABLE_NAME> RENAME COLUMN<OLD_COL_NAME>
TO <NEW_COL_NAME>;

```
alter table customer change city town varchar(20);
desc customer;
alter table customer change column town city varchar(40);
desc customer;
```

DROPING AN EXISTING COLUMN:

Syntax: ALTER TABLE <TABLE_NAME> DROP(COL1_NAME,COL2_NAME);

// drop column/constraint

```
alter table customer drop column city;
alter table customer drop column state, drop column country;
```

```
select * from customer;

alter table customer drop key uk_phone;
alter table emp drop foreign key fk_mgr;
```

DROP: This command is used to remove the table from the database.

```
DROP table customer;
```

TRUNCATE: This command is used to remove all the rows from the table.
TRUNCATE TABLE <TABLE_NAME>;

```
TRUNCATE table customer;
```

// Following statement does not work. Why?

```
truncate emp where deptno = 10;
```

Difference between Delete and Truncate?

Delete	Truncate
We can Roll Back the data	We cannot Roll Back the data
Rows are deleted temporarily	Rows are deleted permanently
Where clause can be used	Where clause cannot be used
Delete is sub language DML	Truncate is sub language DDL

Note: When we use a truncate command the table gets dropped and re-created. As the structure is affected it is called a DDL command.

All DDL commands are permanent.
All DML commands are Temporary.

Rename: This command is used to change the table name.
Syntax: RENAME <OLD_TABLE_NAME> TO <NEW_TABLE_NAME>;

```
rename table emp to employee;
```

Creating duplicate tables or backup tables: By using the combination of create and select, we can create copy of a table.

```
create table emp1 as select * from emp;
create table emp2 as select * from emp where deptno = 30;
create table emp3 as select * from emp where 10;
create table emp4 as select empno, ename, job, deptno from
emp where deptno = 20;

create table emp5 as select * from emp where 1=1;
```

FUNCTIONS: Functions manipulate the data items and gives the result. There are two types of functions.

Aggregate Functions (group functions)

These functions act on an entire set of data and not just on one data element.

Scalar Functions

These functions act only on single data values.

We will look at some examples of how to use aggregate functions.

AVG - The Average Function

The average function returns the average of all the values selected.

```
select AVG(sal) from emp;
```

SUM - The Addition Function

The SUM function returns the sum of all the values selected.

```
select SUM(sal) from emp;
```

MAX - The Maximum Function

The MAX function returns the maximum of all the values selected.

```
select MAX(sal) from emp;
```

MIN - The Minimum Function

The MIN function returns the minimum of all the values selected.

```
select MIN(sal) from emp;
```

COUNT - The Count Function

The count function returns the number of all the values selected.

COUNT(*):

```
select COUNT(*) from emp;
```

COUNT(EXPR): Return number of values present in the column.

```
Select COUNT(sal) from emp;
Select COUNT(empno) from emp;
Select COUNT(comm) from emp;
```

String Scalar functions:

CONCAT: Returns text strings concatenated

```
SELECT CONCAT('Hello', 'World');
SELECT CONCAT('hello', space(3), 'world');
SELECT CONCAT(empno,', ',ename) as 'Employee' from emp;
```

INSTR: Returns the location of a substring in a string.

```
SELECT INSTR('hello' , 'e');
SELECT INSTR('hello' , 't');
```

LENGTH: Returns the number of characters of the specified string expression.

```
SELECT LENGTH('hello');
```

RTRIM: Returns a character string after truncating all trailing blanks.

```
SELECT RTRIM(' hello    ');
```

LTRIM: Returns a character expression after it removes leading blanks.

```
SELECT LTRIM('  hello    ');
SELECT Concat(LTRIM('  hello    '), RTRIM(' hello    '));
SELECT Concat(TRIM('  hello    '), TRIM(' hello    '));
```

REPLACE: Replaces all occurrences of a specified string value with another string value.

```
SELECT REPLACE('hello' , 'e' , '$');
SELECT REPLACE('hello' , 'T' , '$');
```

REVERSE: Returns the reverse order of a string value.

```
SELECT REVERSE('hello');
SELECT REVERSE(concat('hello', 'world'));
```

SUBSTR: Returns part of a text.

```
SELECT SUBSTR('hello',2, 3);
```

LOWER: Returns a character expression after converting uppercase character data to lowercase.

```
SELECT LOWER('HELLO');
```

UPPER: Returns a character expression with lowercase character data converted to uppercase.

```
SELECT UPPER('hello');
```

Date related scalar functions

DATE_ADD: Returns a specified date with additional time values.

```
SELECT DATE_ADD('2008-01-02', INTERVAL 1 DAY);
SELECT DATE_ADD('2008-01-02', INTERVAL 1 WEEK);
SELECT DATE_ADD('2008-01-02', INTERVAL 1 MONTH);
```

DAYOFMONTH: Returns an integer representing the day (day of the month) of the specified date.

```
SELECT DAYOFMONTH('2015-08-30');
```

LAST_DAY: Returns a date representing the last day of the month for specified date.

```
SELECT LAST_DAY('2015-08-02');
SELECT LAST_DAY('2016-02-02');
```

DATEDIFF: Returns the difference between two days, expressed as a value in days.

```
SELECT DATEDIFF('2010-04-01', '2010-03-01');
```

PERIOD_DIFF: returns the number of months between two periods.

```
SELECT PERIOD_DIFF(201005, 201003);
```

SYSDATE(): Returns the current database system date. This value is derived from the operating system of the computer on which the instance of MySQL is running.

```
SELECT SYSDATE();
```

Numeric related scalar functions

FLOOR: Returns an integer that is less than or equal to the specified numeric expression.

```
SELECT FLOOR(59.9);
```

CEIL: Returns an integer that is greater than, or equal to, the specified numeric expression.

```
SELECT CEIL(59.1);
```

ROUND: Returns a numeric value, rounded to the specified length or precision.

```
SELECT ROUND(59.9);
SELECT ROUND(59.1);
```

ABS(): Returns the absolute value of a number.

```
SELECT ABS(-2);
SELECT ABS(3);
```

Conversion scalar functions

DATE_FORMAT: Converts a date into a string

```
SELECT DATE_FORMAT(SYSDATE(), '%Y-%m-%d');
SELECT DATE_FORMAT(SYSDATE(), '%M-%Y-%d');
SELECT DATE_FORMAT(SYSDATE(), '%M-%Y-%D');
SELECT DATE_FORMAT(SYSDATE(), '%M-%y-%d');
```

FORMAT: Converts a number into a string

```
SELECT FORMAT(1003423, 3);
```

CONVERT: Used to convert one datatype into another, may be used to convert a string into a number

```
SELECT CONVERT('11', UNSIGNED INTEGER);
SELECT CONVERT('KMIT', UNSIGNED INTEGER);
```

Null related scalar function

IFNULL: Accepts two arguments and returns the first if it is not NULL

```
SELECT IFNULL(NULL, 'Hello');
SELECT IFNULL('hello', 'world');
SELECT IFNULL(NULL, NULL);
```

Group By clause: Group By clause is used to divide rows into several groups. We can apply aggregate/group function on each group.

```
Select job, avg(sal) from emp Group By job;

Select job, count(*) from emp Group By job;

Select job, count(job) from emp Group By job;

Select Deptno, sum(Sal), min(Sal), max(Sal), avg(Sal),
count(*)
  from emp Group By deptno;
```

We can use the combination of where clause and Group By clause.
First where clause is executed. On the result of where clause, Group By clause is applied.

```
Select deptno, sum(sal) from emp where deptno <> 10 Group
By deptno;

Select deptno, job, sum(sal) from emp Group By deptno, job;
```

Having clause: Having clause is used to filter the output from Group By clause.

```
Select deptno, sum(sal) from emp Group By deptno having
sum(sal) > 6000;
```

Order By clause: Order By clause is used to arrange the rows in the table.
By default order by clause is ascending order. Null values are arranged last.

```
Select * from emp Order By sal;

Select * from emp Order By ename;

select * from emp order by ename ASC;

Select * from emp Order By HIREDATE;

Select * from emp Order By job, sal;

Select * from emp Order By sal DESC;
```

```
select * from emp order by ename DESC, job ASC;
```

```
select * from emp order by ename, job;
```

Note: Order by clause should be the last of the query.

```
Select deptno, sum(sal) from emp Group By deptno Having
sum(sal) > 6000 Order By sum(sal) DESC;
```

```
Select deptno, sum(sal) from emp where ename <> 'Kevin'
Group By deptno;
```

```
Select deptno, sum(sal) from emp where ename <> 'Kevin'
Group By deptno Having sum(sal) > 6000;
```

Integrity Constraints

- Integrity Constraints are the rules or conditions that are imposed on database tables to allow the storage of only legal data into the database for all the legal instances.
- Constraints helps in improving the accuracy and quality of the database.
- We can apply the constraints in two situations on the table.
  - During table creation
  - After table creation

Constraints can be created at two levels

1. Column level
2. Table level

- Column level Constraints: Applying the constraints after defining the column immediately.
- Table level constraints: Applying the constraints after defining all the columns in the table.

There are six types of constraints.

CONSTRAINT	DESCRIPTION

NOT NULL	NOT NULL constraint allows to specify that a column cannot contain any NULL value. NOT NULL can be used to CREATE and ALTER a table.
UNIQUE	The UNIQUE constraint does not allow to insert a duplicate value in a column. The UNIQUE constraint maintains the uniqueness of a column in a table. More than one UNIQUE column can be used in a table
PRIMARY KEY	A PRIMARY KEY constraint for a table enforces the table to accept unique data for a specific column and this constraint creates a unique index for accessing the table faster.
FOREIGN KEY	A FOREIGN KEY creates a link between two tables by one specific column of both table. The specified column in one table must be a PRIMARY KEY and referred by the column of another table known as FOREIGN KEY.
CHECK	A CHECK constraint controls the values in the associated column. The CHECK constraint determines whether the value is valid or not from a logical expression
DEFAULT	Each column must contain a value (including a NULL). While inserting data into a table, if no value is supplied to a column, then the column gets the value set as DEFAULT

<u>Not Null</u>

```
Create table student(Sno integer NOT NULL,
      Sname varchar(10), Marks integer);

insert into student values(101,'Arun',50);
insert into student values(NULL,'Arun',NULL); // Invalid
```

<u>Unique</u>

```
Create table student2(Sno integer Unique,
  collegeId integer unique,
  Sname varchar(10), Marks integer);

insert into student2 values(101,1,'Arun',50);
insert into student2 values(101,2, NULL,50); // Invalid
insert into student2 values(102,2, NULL,50);
insert into student2 values(NULL,3,'Arun',50);
insert into student2 values(NULL,NULL,'Arun',50);
insert into student2 values(NULL,2,'Arun',50); // Invalid
```

UNIQUE constraint can accept multiple null values.

Primary key

A primary key constraint is a combination of NOT NULL and UNIQUE. A primary key constraint does not accept null values as well as duplicate values across the column. Primary key column is used to uniquely identify every row in a table.

Note: A table can have only one primary key.

```
Create table student3(Sno integer Primary key, Sname
      varchar(10), Marks integer);

insert into student3 values(101,'Arun',50);
Insert into student3 values(101,NULL,50);  // Invalid
Insert into student3 values(NULL,'Arun',50); // Invalid
```

Composite Primary key

When primary key is applied to multiple columns it is called composite primary key. Composite primary key can be applied only at table level.

```
Create table student4(firstname varchar(10),
  lastname varchar(10),
  Marks integer,
  PRIMARY KEY(firstname,lastname));

Insert into student4 values('ravi','Reddy',40);
Insert into student4 values('Ravi','Reddy',40); // invalid
Insert into student4 values('Subba','Rao',40);
Insert into student4 values('Kavita','Reddy',40);
Insert into student4 values(NULL,'Arun',40);  // invalid
Insert into student4 values('Raj',NULL,40); // invalid
```

Foreign key constraints or referential integrity

These constraints establish relationship between tables. This relationship is called as parent-child relationship. It is also called master detail relationship.

A foreign key column in the child table will only accept values which are there in the primary key column or unique column of parent table.

Creating parent/master table:

```
Create table school(sno integer, Sname varchar(10), Marks
integer, primary key(sno));

insert into school values(101,'Arun',90);
insert into school values(102,'Fs1',92);
insert into school values(103,'Amit',45);
```

Creating the child/detail table:

```
Create table library(sno integer primary key, Book_name
varchar(10), FOREIGN KEY (sno) REFERENCES school(sno));

Insert into library values(102,'java');
Insert into library values(103,'c++');
Insert into library values(103,'oracle');
Insert into library values(108,'dotnet');
Insert into library values(Null,'DBA');
```

- Foreign key column name need not match with primary key column name or unique column name. But the data type should match.
- To establish relationship, it is mandatory that the parent table should have primary key constraint or at least unique constraints.

What will happen when we execute the following?

```
delete from school where sno = 102;
```

Note: we cannot delete the row from the parent table if the corresponding value exists in child table.

Using on delete cascade:

When we delete the rows from the parent table and then corresponding child table rows are deleted automatically when we use on delete cascade.

```
Create table school1(sno integer, Sname varchar(10), Marks
integer, primary key(sno));

insert into school1 values(101,'Arun',90);
insert into school1 values(102,'Fs1',92);
insert into school1 values(103,'Amit',45);

Create table library1(sno integer primary key, Book_name
varchar(10), FOREIGN KEY (sno) REFERENCES school1(sno) ON
UPDATE CASCADE on delete cascade);

Insert into library1 values(101,'C');
Insert into library1 values(102,'java');
Insert into library1 values(103,'c++');

update school1 set sno=104 where sno=101;
delete from school1 where sno = 102;
```

Check Constraint

```
CREATE TABLE CUSTOMERS(
  ID   INT              NOT NULL,
  NAME VARCHAR (20)     NOT NULL,
  AGE  INT              NOT NULL CHECK (AGE >= 18),
  PRIMARY KEY (ID)
);

Insert into customers values(1, 'Ravi', 25);
Insert into customers values(2, 'Ravi', 15); // not working
```

Default constraint

```
CREATE TABLE CUSTOMERS2(
  ID   INT              NOT NULL,
  NAME VARCHAR (20)     NOT NULL,
  AGE  INT              NOT NULL,
  Country   VARCHAR(30) DEFAULT 'india',
  orderDate TIMESTAMP   DEFAULT now(),
  PRIMARY KEY (ID)
);

insert into customers2 (id, name, age) values (1, "ravi", 25);
insert into customers2 values (2, "subbu", 30, 'usa', now());
insert into customers2 (id, name, age, country) values (3, "ravi", 25, 'USA');
```

Joins

An SQL JOIN clause is used to combine rows from two or more tables, based on a common field between them. Joins can be used to temporarily create 'complete' records from a database which may split related data across several tables.

INNER JOIN: Returns all rows when there is at least one match in BOTH tables

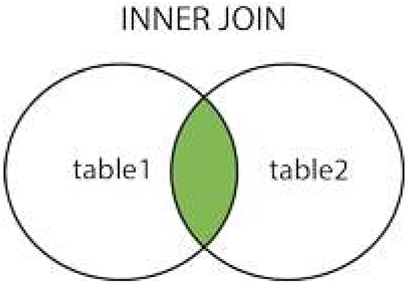
LEFT JOIN: Return all rows from the left table, and the matched rows from the right table

RIGHT JOIN: Return all rows from the right table, and the matched rows from the left table

FULL JOIN: Return all rows when there is a match in ONE of the tables

The most common type of join is: SQL INNER JOIN (simple join). An SQL INNER JOIN returns all rows from multiple tables where the join condition is met.

The INNER JOIN keyword selects all rows from both tables as long as there is a match between the columns. If there are rows in the "emp" table that do not have matches in "dept", these employees will not be listed.

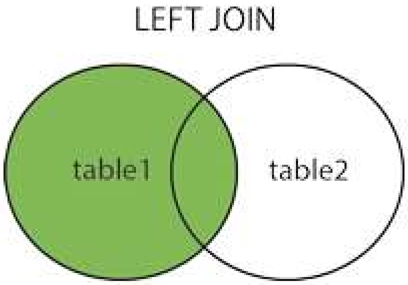


Inner Join and Join are same.

```
select emp.empno, emp.ename, dept.dname, emp.job
from emp
inner join dept
on emp.deptno=dept.deptno;

or

select emp.empno, emp.ename, dept.dname, emp.job
from emp
join dept
on emp.deptno=dept.deptno;

or

select e.empno, e.ename, d.dname, e.job
from emp as e, dept as d
where e.deptno=d.deptno;
```

The LEFT JOIN keyword returns all rows from the left table (table1), with the matching rows in the right table (table2). The result is NULL in the right side when there is no match.

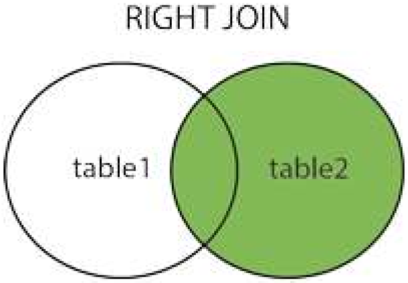
Note: The LEFT JOIN keyword returns all the rows from the left table (emp), even if there are no matches in the right table (dept).



```
SELECT emp.empno, emp.ename, dept.dname, emp.job
FROM emp
LEFT JOIN dept
ON emp.deptno=dept.deptno;
```

The RIGHT JOIN or RIGHT OUTER JOIN keyword returns all rows from the right table (table2), with the matching rows in the left table (table1).
The result is NULL in the left side when there is no match.

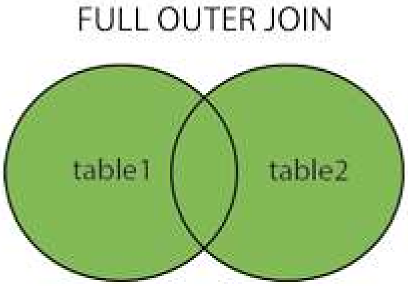
Note: The RIGHT JOIN keyword returns all the rows from the right table (dept), even if there are no matches in the left table (emp).



```
SELECT emp.empno, emp.ename, dept.dname, emp.job
FROM emp
RIGHT JOIN dept
ON emp.deptno=dept.deptno;
```

The FULL OUTER JOIN keyword returns all rows from the left table (table1) and from the right table (table2).
The FULL OUTER JOIN keyword combines the result of both LEFT and RIGHT joins.

MySQL doesn't support this. We can use union to achieve the same.



The UNION operator is used to combine the result-set of two or more SELECT statements.

Notice that each SELECT statement within the UNION must have the same number of columns. The columns must also have similar data types. Also, the columns in each SELECT statement must be in the same order.

```
SELECT * FROM emp
LEFT JOIN dept ON emp.deptno=dept.deptno
UNION
SELECT * FROM emp
RIGHT JOIN dept ON emp.deptno=dept.deptno;
```

Union All (duplicate values also)

```
SELECT * FROM emp LEFT JOIN dept ON emp.deptno=dept.deptno
union all
SELECT * FROM emp RIGHT JOIN dept ON
emp.deptno=dept.deptno;
```

SELF JOIN: When a table is joining to itself it is called self - join. In self joins we need to create two table aliases for the same table. Whenever we have relationship between two columns in the same table then we need to use self-join.

```
select e1.empno, e2.mgr from emp e1,emp e2 where
e1.empno=e2.mgr;
```

Cartesian product:

When tables are joined without any join condition it is called Cartesian product. In the result we get all possible combination.

```
Select e.empno, e.ename, e.sal, e.deptno, d.dname from emp
e, dept d;
```

<u>Cross product/cross join</u>: It is same as Cartesian product.

```
Select e.empno, e.ename, e.sal, e.deptno, d.dname from   emp
as e CROSS JOIN dept as d;
```

What does the following query return?

```
SELECT *
FROM dept
WHERE EXISTS (SELECT *
              FROM emp
              WHERE emp.deptno = dept.deptno);
```

```
SELECT *
FROM dept
WHERE NOT EXISTS (SELECT *
              FROM emp
              WHERE emp.deptno = dept.deptno);
```

<u>TCL</u> <u>(Transaction Control Language</u>): It is collection of three commands. They are

1. COMMIT - make changes permanent to the database.
2. ROLLBACK - the changes which are not permanent.
3. SAVE POINT - is logical marking given for series of transactions. Instead of rollback completely, we can rollback to a save point.

```
Use test;

START TRANSACTION;

insert into emp values
(8149,'Ravi','Manager',7839,'93/6/13',800,0.00,20);
```

```
SELECT * FROM emp;

SAVEPOINT tran1;

insert into emp values
(8343,'Teja','Manager',7839,'93/6/13',800,0.00,20);

SELECT * FROM emp;

SAVEPOINT tran2;

ROLLBACK TO tran2;

SELECT * FROM emp;
ROLLBACK TO tran1;
SELECT * FROM emp;
```

Note: Update last rollback to commit and test.

Stored procedures

A stored procedure is a segment of declarative SQL statements stored inside the database catalog. A stored procedure can be invoked by triggers, other stored procedures, and applications such as VB, C++, Java, Python, PHP, etc.

Putting database-intensive operations into stored procedures lets you define an API for your database application. You can reuse this API across multiple applications and multiple programming languages. This technique avoids duplicating database code, saving time and effort when you make updates due to schema changes, tune the performance of queries, or add new database operations for logging, security etc.

Stored procedures advantages

- Typically stored procedures help increase the performance of the applications. Once created, stored procedures are compiled and stored in the database. However, MySQL implements the stored procedures slightly different. MySQL stored procedures are compiled on demand. After compiling a stored procedure, MySQL puts it into a cache, and MySQL maintains its own stored procedure cache for every single connection. If an application uses a stored procedure multiple times in a single connection, the compiled version is used; otherwise, the stored procedure works like a query.

- Stored procedures help reduce the traffic between application and database server because instead of sending multiple lengthy SQL statements, the application has to send only name and parameters of the stored procedure.

- Stored procedures are reusable and transparent to any applications. Stored procedures expose the database interface to all applications so that developers don't have to develop functions that are already supported in stored procedures.

- Stored procedures are secure. The database administrator can grant appropriate permissions to applications that access stored procedures in the database without giving any permission on the underlying database tables.

MySQL stored procedures disadvantages

- If you use a lot of stored procedures, the memory usage of every connection that is using those stored procedures will increase substantially. In addition, if you overuse a large number of logical operations inside store procedures, the CPU usage will also increase because the database server is not well-designed for logical operations.

- Constructs of stored procedures make it more difficult to develop stored procedures that have complicated business logic.

- It is difficult to debug stored procedures. Only a few database management systems allow you to debug stored procedures. Unfortunately, MySQL does not provide facilities for debugging stored procedures.

- It is not easy to develop and maintain stored procedures. Developing and maintaining stored procedures are often required a specialized skill set that not all application developers possess. This may lead to problems in both application development and maintenance phases.

Most stored procedures that we write require parameters. The parameters make the stored procedure more flexible and useful. In MySQL, a parameter has one of three modes: IN, OUT, or INOUT.

IN – is the default mode. When you define an IN parameter in a stored procedure, the calling program has to pass an argument to the stored procedure. In addition, the value of an IN parameter is protected. It means that even the value of the IN parameter is changed inside the stored procedure, its original value is retained after the stored procedure ends. In other words, the stored procedure only works on the copy of the IN parameter.

OUT – the value of an OUT parameter can be changed inside the stored procedure and its new value is passed back to the calling program. Notice that the stored procedure cannot access the initial value of the OUT parameter when it starts.

INOUT – an INOUT parameter is the combination of IN and OUT parameters. It means that the calling program may pass the argument, and the stored procedure can modify the INOUT parameter and pass the new value back to the calling program.

```
use test;

# ----------------------------------------
DROP PROCEDURE IF EXISTS empCount;

# Get count of employees
CREATE PROCEDURE empCount (OUT param1 INT)
SELECT COUNT(*) INTO param1 FROM emp;

# calling/invoking the procedure
CALL empCount(@num);
select @num;

# ------------------------------------------
DROP PROCEDURE IF EXISTS empAverageSal;

# Get average salary of employees

DELIMITER $$
CREATE PROCEDURE empAverageSal(OUT avgSal decimal(10,2))
```

```
 BEGIN
 SELECT AVG(sal) INTO avgSal FROM emp;
 END $$

DELIMITER ;

call empAverageSal(@sal);
select @sal;

# -------------------------------------------
DROP PROCEDURE IF EXISTS getJob;

# Given empno, get the job
CREATE PROCEDURE getJob (IN param1 INT, OUT param2 INT)
SELECT job from emp where empno = param1;

CALL getJob(7788, @job);
select @job;

# -------------------------------------------
DROP PROCEDURE IF EXISTS setCounter;

DELIMITER $$
CREATE  PROCEDURE  setCounter(INOUT  count  INT(3),IN  inc
INT(3))
BEGIN
 SET count = count + inc;
END$$
DELIMITER ;

SET @counter = 1;
CALL setCounter(@counter,1);
CALL setCounter(@counter,2);
CALL setCounter(@counter,3);
SELECT @counter;

# -------------------------------------------
DROP PROCEDURE IF EXISTS getEmpSalGrade;
```

```
DELIMITER $$

CREATE PROCEDURE getEmpSalGrade(
    in  p_empNo int,
    out p_empSalGrade  varchar(10))
BEGIN
    DECLARE empSal double;

    SELECT sal INTO empSal
    FROM emp
    WHERE empno = p_empNo;

    IF empSal > 3000 THEN
      SET p_empSalGrade = 'HIGH';
    ELSEIF (empSal > 1000 && empSal <= 3000 ) THEN
      SET p_empSalGrade = 'MEDIUM';
    ELSEIF (empSal <= 1000) THEN
      SET p_empSalGrade = 'LOW';
    END IF;

END$$

DELIMITER ;

call getEmpSalGrade(7839, @empSalGrade);
select @empSalGrade;

call getEmpSalGrade(7369, @empSalGrade);
select @empSalGrade;

call getEmpSalGrade(7698, @empSalGrade);
select @empSalGrade;

# ------------------------------------------
DROP PROCEDURE IF EXISTS getEmpSalGrade2;
DELIMITER $$

CREATE PROCEDURE getEmpSalGrade2(
    in  p_empNo int,
    out p_empSalGrade  varchar(10))
BEGIN
    DECLARE empSal double;

    SELECT sal INTO empSal
    FROM emp
    WHERE empno = p_empNo;

    CASE
    WHEN empSal > 3000 THEN
      SET p_empSalGrade = 'HIGH';
    WHEN (empSal > 1000 && empSal <= 3000 ) THEN
      SET p_empSalGrade = 'MEDIUM';
    WHEN (empSal <= 1000) THEN
      SET p_empSalGrade = 'LOW';
    END CASE;

END$$

DELIMITER ;

call getEmpSalGrade2(7839, @empSalGrade);
select @empSalGrade;

call getEmpSalGrade2(7369, @empSalGrade);
select @empSalGrade;

call getEmpSalGrade2(7698, @empSalGrade);
select @empSalGrade;

# ------------------------------------------
DROP PROCEDURE IF EXISTS setShippingDays;
DELIMITER $$

CREATE PROCEDURE setShippingDays(
    in  p_deptNo int,
    out p_shippingDays  varchar(50))
```

```
BEGIN
    DECLARE cLocation varchar(50);

    SELECT location INTO cLocation
    FROM dept
    WHERE deptno = p_deptNo;

    CASE cLocation
    when  'Dallas' THEN
      SET p_shippingDays = '1 day shipping';
    when  'Tempe' THEN
      SET p_shippingDays = '2 day2 shipping';
    when  'Chicago' THEN
      SET p_shippingDays = '3 days shipping';
    ELSE
      SET p_shippingDays = '7 days shipping';
    END CASE;

END$$

DELIMITER ;

call setShippingDays(20, @custShippingDays);
select @custShippingDays;

call setShippingDays(50, @custShippingDays);
select @custShippingDays;

call setShippingDays(30, @custShippingDays);
select @custShippingDays;
```

Stored function

A stored function is a special kind of stored program that returns a single value. You use stored functions to encapsulate common formulas or business rules that are reusable among SQL statements or stored programs.

First, you specify the name of the stored function after CREATE FUNCTION clause.

Second, you list all parameters of the stored function inside the parentheses. By default, all parameters are IN parameters. You cannot specify IN, OUT or INOUT modifiers to the parameters.

Third, you must specify the data type of the return value in the RETURNS statement. It can be any valid MySQL data types.

Fourth, you write the code in the body of the stored function. It can be a single statement or a compound statement. Inside the body section, you have to specify at least one RETURN statement. The RETURN statement returns a value to the caller. Whenever the RETURN statement is reached, the stored function's execution is terminated immediately.

```
# ----------------------------------------
use test;

DROP function IF EXISTS getSalGrade;
DELIMITER $$

CREATE function getSalGrade(p_empNo int) returns varchar(10)
BEGIN
    DECLARE empSal double;
    DECLARE empSalGrade varchar(10);

    SELECT sal INTO empSal
    FROM emp
    WHERE empno = p_empNo;

    IF empSal > 3000 THEN
      SET empSalGrade = 'HIGH';
    ELSEIF (empSal > 1000 && empSal <= 3000 ) THEN
      SET empSalGrade = 'MEDIUM';
    ELSEIF (empSal <= 1000) THEN
      SET empSalGrade = 'LOW';
    END IF;

    return empSalGrade;
```

```
END$$

select sal, getSalGrade(7839) from emp where empno = 7839;
select sal, getSalGrade(7369) from emp where empno = 7369;
select sal, getSalGrade(7698) from emp where empno = 7698;
```

Note:

1) Query to retrieve MySQL tables

```
select TABLE_NAME from information_schema.TABLES where
TABLE_SCHEMA = 'mysql';
```

2) To get the list of all procedures and functions execute the following sql statement:

```
select name, type from mysql.proc where db = database()
order by type, name;
```

3) Query to get list of users in mysql DB.

```
select * from mysql.user;

select distinct user from mysql.user where user <> '' order
by user;
```

4) Query to get details of stored procedure and function.

```
show create procedure setCounter;
show create function getSalGrade;
```

Triggers

A SQL trigger is a set of SQL statements stored in the database catalog. A SQL trigger is executed or fired whenever an event associated with a table occurs e.g insert, update or delete.

A SQL trigger is a special type of stored procedure. It is special because it is not called directly like a stored procedure. The main difference between a trigger and a stored procedure is that a trigger is called automatically when a data modification event is made against a table whereas a stored procedure must be called explicitly.

Advantages of using SQL triggers

- SQL triggers provide an alternative way to check the integrity of data.
- SQL triggers can catch errors in business logic in the database layer.
- SQL triggers provide an alternative way to run scheduled tasks. By using SQL triggers, you don't have to wait to run the scheduled tasks because the triggers are invoked automatically before or after a change is made to the data in the tables.
- SQL triggers are very useful to audit the changes of data in tables.

Disadvantages of using SQL triggers

- SQL triggers only can provide an extended validation and they cannot replace all the validations. Some simple validations have to be done in the application layer. For example, you can validate user's inputs in the client side by using JavaScript or in the server side using server-side scripting languages such as JSP, PHP, ASP.NET, Perl, etc.
- SQL triggers are invoked and executed invisible from the client applications; therefore, it is difficult to figure out what happen in the database layer.
- SQL triggers may increase the overhead of the database server.

```
CREATE TRIGGER trigger_name trigger_time trigger_event
on table_name
for each row
Begin

End;
```

You put the trigger name after the CREATE TRIGGER statement. The trigger name should follow the naming convention [trigger time]_[table name]_[trigger event], for example before_emp_update.

Trigger activation time can be BEFORE or AFTER. You must specify the activation time when you define a trigger. You use the BEFORE keyword if you want to process action prior to the change is made on the table and AFTER if you need to process action after the change is made.

The trigger event can be INSERT, UPDATE or DELETE. This event causes the trigger to be invoked. A trigger only can be invoked by one event. To define a trigger that is invoked by multiple events, you have to define multiple triggers, one for each event.

A trigger must be associated with a specific table. Without a table trigger would not exist therefore you have to specify the table name after the ON keyword.

You place the SQL statements between BEGIN and END block. This is where you define the logic for the trigger.

```
# ----------------------------------------

use test;

DROP TABLE IF EXISTS emp_audit;

CREATE TABLE emp_audit (
    id INT AUTO_INCREMENT PRIMARY KEY,
    empno INT NOT NULL,
    ename VARCHAR(50) NOT NULL,
    changedat DATETIME DEFAULT NULL,
    action VARCHAR(50) DEFAULT NULL
);

DROP TRIGGER IF EXISTS before_emp_update;

DELIMITER $$
CREATE TRIGGER before_emp_update
    BEFORE UPDATE ON emp
    FOR EACH ROW
BEGIN
    INSERT INTO emp_audit
    SET action = 'Before update',
      empno = OLD.empno,
      ename = OLD.ename,
      changedat = NOW();
END$$
DELIMITER ;

update emp set sal = '5000' where empno = 7876;
update emp set job = 'analyst', deptno = 30 where empno = 7876;

select * from emp_audit;

# ---------------------------------------------

DROP TRIGGER IF EXISTS oninsert_emp;

DELIMITER $$
CREATE TRIGGER oninsert_emp
    AFTER INSERT ON emp
    FOR EACH ROW
BEGIN
    INSERT INTO emp_audit
    SET action = 'After insert',
      empno = NEW.empno,
      ename = NEW.ename,
      changedat = NOW();
END$$
DELIMITER ;

insert into emp values
(9999,'David','Manager',7902,'93/6/13',800,0.00,20);
insert into emp values
(8888,'David','Manager',7902,'93/6/13',800,0.00,20);

select * from emp_audit;
# ---------------------------------------------

DROP TRIGGER IF EXISTS ondelete_emp;

DELIMITER $$
CREATE TRIGGER ondelete_emp
```

```
    AFTER DELETE ON emp
    FOR EACH ROW
BEGIN
    INSERT INTO emp_audit
    SET action = 'After delete',
      empno = OLD.empno,
      ename = OLD.ename,
      changedat = NOW();
END$$
DELIMITER ;

delete from emp where empno = 9999;
delete from emp where empno = 8888;

select * from emp_audit;
```

Note: To get the list of all triggers execute the following sql statement:

```
select trigger_name FROM information_schema.TRIGGERS WHERE
TRIGGER_SCHEMA=database();

SELECT * FROM information_schema.TRIGGERS WHERE
TRIGGER_SCHEMA=database();

show create trigger ondelete_emp;
```

MySQL truncates the table by dropping and creating the table. Thus, the DELETE triggers for the table do not fire during the truncation.

Views

A database view is known as a "virtual table" that allows you to query the data in it. MySQL views are not only queryable but also updatable.

A view is a logical snapshot based on a table or another view. It is used for -

Restricting access to data;
Making complex queries simple;

Ensuring data independency;
Providing different views of same data.

```
# -----------------------------------------

use test;

DROP VIEW IF EXISTS empDetails;

CREATE VIEW empDetails AS
SELECT
    empno, ename, job, deptno, count(job) as total
FROM
    emp
GROUP by job
ORDER BY ename;

select * from empDetails;

# -----------------------------------------

DROP VIEW IF EXISTS empDept;

CREATE or REPLACE VIEW empDept AS
  select emp.empno, emp.ename, dept.dname, emp.job
  from emp
  inner join dept
  on emp.deptno=dept.deptno
  ORDER BY ename;

select * from empDept;

-- Show Create View empDept;

Alter view empDept
As
  select emp.empno, emp.ename, dept.dname
  from emp
  inner join dept
  on emp.deptno=dept.deptno;

select * from empDept;
```

Note: To get the list of all views execute the following sql statement:

```
show full tables where table_type = 'view';

show create view empDept;
```