

II.....SQL - Constraints

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

Constraints could be either on a column level or a table level. The column level constraints are applied only to one column, whereas the table level constraints are applied to the whole table.

Following are some of the most commonly used constraints available in SQL. These constraints have already been discussed in SQL - RDBMS Concepts chapter, but it's worth to revise them at this point.

- NOT NULL Constraint – Ensures that a column cannot have NULL value.
- DEFAULT Constraint – Provides a default value for a column when none is specified.
- UNIQUE Constraint – Ensures that all values in a column are different.
- PRIMARY Key – Uniquely identifies each row/record in a database table.

- FOREIGN Key – Uniquely identifies a row/record in any of the given database table.
- CHECK Constraint – The CHECK constraint ensures that all the values in a column satisfies certain conditions.
- INDEX – Used to create and retrieve data from the database very quickly.

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

Dropping Constraints

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

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

```
ALTER TABLE EMPLOYEES DROP CONSTRAINT EMPLOYEES_PK;
```

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

```
ALTER TABLE EMPLOYEES DROP PRIMARY KEY;
```

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

Integrity Constraints

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

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

SQL - NOT NULL Constraint

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

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

Example

For example, the following SQL query creates a new table called CUSTOMERS and adds five columns, three of which, are ID NAME and AGE, In this we specify not to accept NULLs –

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

If CUSTOMERS table has already been created, then to add a NOT NULL constraint to the SALARY column in Oracle and MySQL, you would write a query like the one that is shown in the following code block.

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

SQL - DEFAULT Constraint

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

Example

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

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

If the CUSTOMERS table has already been created, then to add a DEFAULT constraint to the SALARY column, you would write a query like the one which is shown in the code block below.

```
ALTER TABLE CUSTOMERS
```

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

Drop Default Constraint

To drop a DEFAULT constraint, use the following SQL query.

```
ALTER TABLE CUSTOMERS  
  
    ALTER COLUMN SALARY DROP DEFAULT;
```

SQL - UNIQUE Constraint

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

Example

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

```
CREATE TABLE CUSTOMERS (
```

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

If the CUSTOMERS table has already been created, then to add a UNIQUE constraint to the AGE column. You would write a statement like the query that is given in the code block below.

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

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

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


DROP a UNIQUE Constraint

To drop a UNIQUE constraint, use the following SQL query.

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

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

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

SQL PRIMARY KEY

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

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

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

In designing the composite primary key, you should use as few columns as possible. It is good for storage and performance both, the more columns you use for primary key the more storage space you require.

Inn terms of performance, less data means the database can process faster.

Points to remember for primary key:

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

When we specify a primary key constraint for a table, database engine automatically creates a unique index for the primary key column.

Main advantage of primary key:

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

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

SQL primary key for one column:

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

MySQL:

1. **CREATE TABLE** students
2. (
3. S_Id **int** NOT NULL,
4. LastName **varchar** (255) NOT NULL,
5. FirstName **varchar** (255),
6. Address **varchar** (255),
7. City **varchar** (255),
8. **PRIMARY KEY** (S_Id)
9.)

SQL Server, Oracle, MS Access:

1. **CREATE TABLE** students

```
2.(
3.S_Id int NOT NULL PRIMARY KEY,
4.LastName varchar (255) NOT NULL,
5.FirstName varchar (255),
6.Address varchar (255),
7.City varchar (255),
8.)
```

SQL primary key for multiple columns:

MySQL, SQL Server, Oracle, MS Access:

```
1.CREATE TABLE students
2.(
3.S_Id int NOT NULL,
4.LastName varchar (255) NOT NULL,
5.FirstName varchar (255),
6.Address varchar (255),
7.City varchar (255),
8.CONSTRAINT pk_StudentID PRIMARY KEY (S_Id, LastName)
9.)
```

Note: you should note that in the above example there is only one PRIMARY KEY (pk_StudentID). However it is made up of two columns (S_Id and LastName).

SQL primary key on ALTER TABLE

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

Primary key on one column:

1. **ALTER TABLE** students
2. **ADD PRIMARY KEY** (S_Id)

Primary key on multiple column:

1. **ALTER TABLE** students
2. **ADD CONSTRAINT** pk_StudentID **PRIMARY KEY** (S_Id,LastName)

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

How to DROP a PRIMARY KEY constraint?

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

MySQL:

1. **ALTER TABLE** students
2. **DROP PRIMARY KEY**

SQL Server / Oracle / MS Access:

1. **ALTER TABLE** students
2. **DROP CONSTRAINT** pk_StudentID

SQL FOREIGN KEY

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

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

primary key in another table.

Let us take an example to explain it:

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

Here orders are given by students.

First table:

S_Id	LastName	FirstName
1	MAURYA	AJEET
2	JAISWAL	RATAN
3	ARORA	SAUMYA

Second table:

O_Id	OrderNo
1	99586465
2	78466588
3	22354846
4	57698656

Here you see that "S_Id" column in the "Orders" table points to the "S_Id" column in

"Students" table.

- The "S_Id" column in the "Students" table is the PRIMARY KEY in the "Students" table.
- The "S_Id" column in the "Orders" table is a FOREIGN KEY in the "Orders" table.

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

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

SQL FOREIGN KEY constraint ON CREATE TABLE:

(Defining a foreign key constraint on single column)

To create a foreign key on the "S_Id" column when the "Orders" table is created:

MySQL:

1. **CREATE TABLE** orders
2. (
3. O_Id **int** NOT NULL,
4. Order_No **int** NOT NULL,
5. S_Id **int**,
6. PRIMARY **KEY** (O_Id),
7. **FOREIGN KEY** (S_Id) **REFERENCES** Persons (S_Id)
8.)

SQL Server / Oracle / MS Access:

1. **CREATE TABLE** Orders

```
2.(  
3.O_Id int NOT NULL PRIMAY KEY,  
4.Order_No int NOT NULL,  
5.S_Id int FOREIGN KEY REFERENCES persons (S_Id)  
6.)
```

SQL FOREIGN KEY constraint for ALTER TABLE:

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

Defining a foreign key constraint on single column:

MySQL / SQL Server / Oracle / MS Access:

```
1.ALTER TABLE Orders  
2.ADD CONSTRAINT fk_PerOrders  
3.FOREIGN KEY(S_Id)
```

4. **REFERENCES** Students (S_Id)

DROP SYNTAX for FOREIGN KEY COSTRAINT:

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

MySQL:

1. **ALTER TABLE** Orders
2. **ROP FOREIGN KEY** fk_PerOrders

SQL Server / Oracle / MS Access:

1. **ALTER TABLE** Orders
2. **DROP CONSTRAINT** fk_PerOrders

Difference between primary key and foreign key in SQL:

These are some important difference between primary key and foreign key

in SQL-

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

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

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

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

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

SQL Composite Key

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

Sometimes more than one attributes are needed to uniquely identify an entity. A primary key that is made by the combination of more than one attribute is known as a composite key.

In other words we can say that:

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

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

SQL Syntax to specify composite key:

1. **CREATE TABLE** TABLE_NAME
2. (COLUMN_1, DATA_TYPE_1,
3. COLUMN_2, DATA_TYPE_2,
4. ???

5. **PRIMARY KEY** (COLUMN_1, COLUMN_2, ...));

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

MySQL:

1. **CREATE TABLE** SAMPLE_TABLE

2. (COL1 **integer**,

3. COL2 **varchar**(30),

4. COL3 **varchar**(50),

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

SQL INDEXES

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

For example, if you want to reference all pages in a book that 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.

The difference between a KEY and an INDEX in MySQL

KEY and INDEX are synonyms in MySQL. They mean the same thing. In databases you would use indexes to improve the speed of data retrieval. An index is typically created on columns used in JOIN, WHERE, and ORDER BY clauses.

Imagine you have a table called users and you want to search for all the users which have the last name 'Smith'. Without an index, the database would have to go through all the records of the table: this is slow, because the more records you have in your database, the more work it has to do to find the result. On the other hand, an index will help the database skip quickly to the

relevant pages where the 'Smith' records are held. This is very similar to how we, humans, go through a phone book directory to find someone by the last name: We don't start searching through the directory from cover to cover, as long we inserted the information in some order that we can use to skip quickly to the 'S' pages.

Primary keys and unique keys are similar. A primary key is a column, or a combination of columns, that can uniquely identify a row. It is a special case of unique key. A table can have at most one primary key, but more than one unique key. When you specify a unique key on a column, no two distinct rows in a table can have the same value.

Also note that columns defined as primary keys or unique keys are automatically indexed in MySQL.

So

All tables must have a primary key, which is used to uniquely identify each of the relation's entries. It may also have foreign keys, which link tables together.

Indexes are used to make searching by keys faster, and to help enforce constraints. Constraints might be used to ensure that the primary key is unique, or that the foreign key references a valid entry in its target table.

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

The basic syntax of a **CREATE INDEX** is as follows.

```
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);
```

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

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

Implicit Indexes

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

The DROP INDEX Command

An index can be dropped using SQL **DROP** command. Care should be taken when dropping an index because the performance may either slow down or improve.

The basic syntax is as follows –

```
DROP INDEX index_name;
```

You can check the [INDEX Constraint](#) chapter to see some actual examples on Indexes.

When should indexes be avoided?

Although indexes are intended to enhance a database's performance, there are times when they should be avoided.

The following guidelines indicate when the use of an index should be reconsidered.

- Indexes should not be used on small tables.
- Tables that have frequent, large batch 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 - CHECK Constraint

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

Example

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

```
CREATE TABLE CUSTOMERS (
```

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

If the CUSTOMERS table has already been created, then to add a CHECK constraint to AGE column, you would write a statement like the one given below.

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

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

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

DROP a CHECK Constraint

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

```
ALTER TABLE CUSTOMERS  
    DROP CONSTRAINT myCheckConstraint;
```

SQL JOINS

As the name shows, JOIN means *to combine something*. In case of SQL, JOIN means **"to combine two or more tables"**.

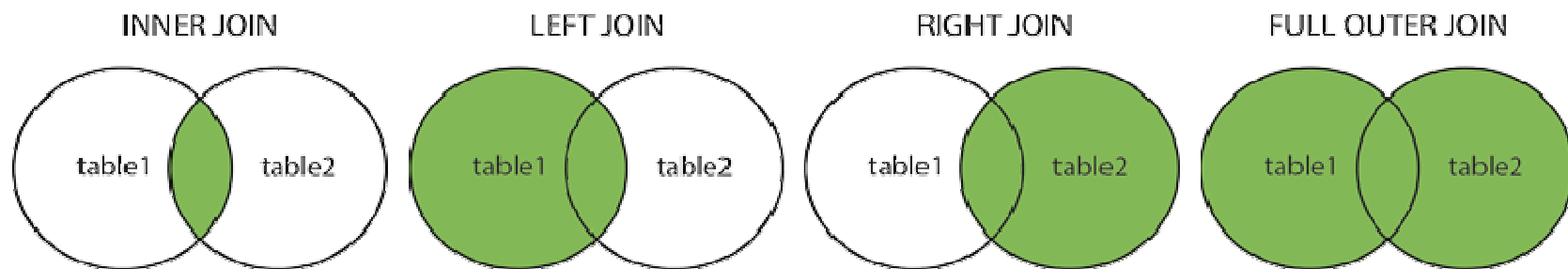
The SQL JOIN clause takes records from two or more tables in a database and combines it together.

ANSI standard SQL defines five types of JOIN :

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

5.cross join.

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



Why SQL JOIN is used?

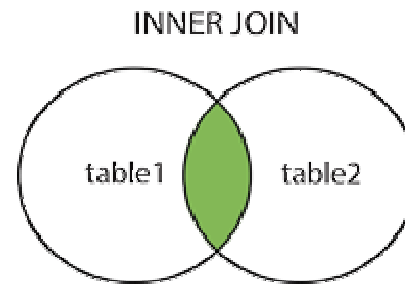
If you want to access more than one table through a select statement.

If you want to combine two or more table then SQL JOIN statement is used .it combines rows of that tables in one table and one can retrieve the information by a SELECT statement.

The joining of two or more tables is based on common field between them.

SQL INNER JOIN also known as simple join is the most common type of join.

SQL INNER JOIN



How to use SQL join or SQL Inner Join?

Let an example to deploy SQL JOIN process:

1.Staff table

ID	Staff_NAME	Staff_AGE	STAFF_ADDRESS	Monthley_Package
1	ARYAN	22	MUMBAI	18000
2	SUSHIL	32	DELHI	20000

3	MONTY	25	MOHALI	22000
4	AMIT	20	ALLAHABAD	12000

2.Payment table

Payment_ID	DATE	Staff_ID	AMOUNT
101	30/12/2009	1	3000.00
102	22/02/2010	3	2500.00
103	23/02/2010	4	3500.00

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

```
SELECT Staff_ID, Staff_NAME, Staff_AGE, AMOUNT
FROM STAFF s, PAYMENT p
WHERE s.ID =p.STAFF_ID;
```

This will produce the result like this:

STAFF_ID	NAME	Staff_AGE	AMOUNT
3	MONTY	25	2500
1	ARYAN	22	3000
4	AMIT	25	3500

SQL OUTER JOIN

In the SQL outer JOIN all the content of the both tables are integrated together either they are matched or not.

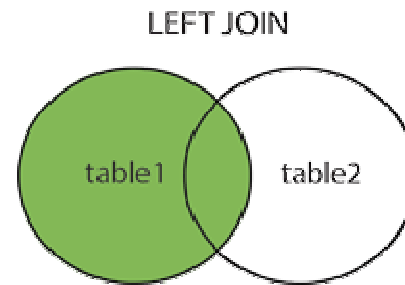
If you take an example of employee table

Outer join of two types:

1.Left outer join (also known as left join): this join returns all the rows from left table combine with the matching rows of the right table. If you get no matching in the right table it returns NULL values.

2.Right outer join (also known as right join): this join returns all the rows from right table are combined with the matching rows of left table .If you get no column matching in the left table .it returns null value.

SQL LEFT JOIN



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

BASIC SYNTAX FOR LEFT JOIN:

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

FROM table1

LEFT JOIN table2

ON table1.column_field = table2.column_field;

CUSTOMER TABLE:

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

This is second table

ORDER TABLE:

O_ID	DATE	CUSTOMER_ID	AMOUNT
001	20-01-2012	2	3000
002	12-02-2012	2	2000
003	22-03-2012	3	4000
004	11-04-2012	4	5000

join these two tables with LEFT JOIN:

SQL

SELECT ID, NAME, AMOUNT,DATE

FROM CUSTOMER

LEFT JOIN ORDER

ON CUSTOMER.ID = ORDER.CUSTOMER_ID;

This will produce the following result:

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

This will list all customers, whether they placed any order or not.
The ORDER BY TotalAmount shows the customers without orders first (i.e. TotalMount is NULL).

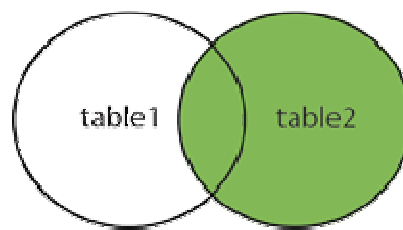
Results: 832 records

OrderNumber	TotalAmount	FirstName	LastName	City	Country
-------------	-------------	-----------	----------	------	---------

NULL	NULL	Diego	Roel	Madrid	Spain
NULL	NULL	Marie	Bertrand	Paris	France
542912	12.50	Patricio	Simpson	Buenos Aires	Argentina
542937	18.40	Paolo	Accorti	Torino	Italy
542897	28.00	Pascale	Cartrain	Charleroi	Belgium
542716	28.00	Maurizio	Moroni	Reggio Emilia	Italy
543028	30.00	Yvonne	Moncada	Buenos Aires	Argentina
543013	36.00	Fran	Wilson	Portland	USA

SQL RIGHT JOIN

RIGHT JOIN



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

Basic syntax for right join:

```
SELECT table1.column1, table2.column2.....
```

```
FROM table1
```

```
RIGHT JOIN table2
```

```
ON table1.column_field = table2.column_field;
```

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

CUSTOMER TABLE:

ID	NAME	AGE	SALARY
1	ARYAN	51	56000

2	AROHI	21	25000
3	VINEET	24	31000
4	AJEET	23	32000
5	RAVI	23	42000

and this is the second table:

ORDER TABLE:

DATE	O_ID	CUSTOMER_ID	AMOUNT
20-01-2012	001	2	3000
12-02-2012	002	2	2000
22-03-2012	003	3	4000
11-04-2012	004	4	5000

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

```
SELECT ID,NAME,AMOUNT,DATE  
FROM CUSTOMER  
RIGHT JOIN ORDER  
ON CUSTOMER.ID = ORDER.CUSTOMER_ID;
```

////////

DEPARTMENT AND EMP
RIGHT OUTER JOIN EG

LIST OF ALL STUDENTS WHETHER DEPT IS ASSIGNED OR NOT

DEPT

1 HR

2 FINANCE

3 DEV

EMPLOYEE

1 PARIMAL 1

2 ANIKET 1

3 VIDYABHUSHAN 2

4 SHUBHAM

5 PRANAV

SELECT EMP.NAME, DEP.NAME

FROM DEPT DEP

RIGHT JOIN

EMPLOYEE EMP

ON DEP.ID = EMP.DEPT_ID;

PARIMAL HR

ANIKET HR

VIDYABHUSHAN FINANCE

SHUBHAM NULL

PRANAV NULL

if innert join

PARIMAL HR

ANIKET HR

VIDYABHUSHAN FINANCE

IF LEFT JOIN

HR PARIMAL

HR ANIKET

FINANCE VIDYABHUSHAN

////////

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

SQL RIGHT JOIN Example

Problem: List customers that have not placed orders

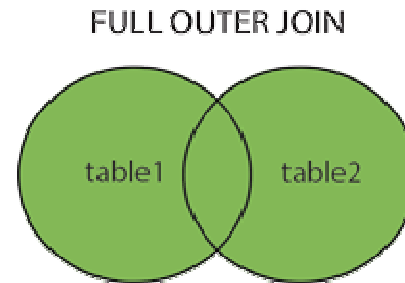
```
1.      SELECT TotalAmount, FirstName, LastName, City, Country
2.      FROM [Order] O RIGHT JOIN Customer C
3.      ON O.CustomerId = C.Id
4.      WHERE TotalAmount IS NULL
```

This returns customers that, when joined, have no matching order.

Results: 2 records

TotalAmount	FirstName	LastName	City	Country
NULL	Diego	Roel	Madrid	Spain
NULL	Marie	Bertrand	Paris	France

SQL FULL OUTER JOIN



What is SQL full outer join?

SQL full outer join is used to combine the result of both left and right outer join and returns all rows (doesn't care its matched or unmatched) from the both participating tables.

Syntax for full outer join:

```
SELECT *
```


FROM table1

FULL OUTER JOIN table2

ON table1.column_name = table2.column_name;

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

- FULL JOIN returns all matching records from both tables whether the other table matches or not.
- FULL JOIN can potentially return very large datasets.
- FULL JOIN and FULL OUTER JOIN are the same.

Let us take two tables to demonstrate full outer join:

table_A

A	M
1	m

2	n
4	o

table_B

A	N
2	p
3	q
5	r

Resulting table

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

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

SQL FULL JOIN Examples

Problem: Match all customers and suppliers by country

```
1.      SELECT C.FirstName, C.LastName, C.Country AS CustomerCountry,
2.          S.Country AS SupplierCountry, S.CompanyName
3.      FROM Customer C FULL JOIN Supplier S
4.          ON C.Country = S.Country
5.      ORDER BY C.Country, S.Country
```

This returns suppliers that have no customers in their country,
and customers that have no suppliers in their country,
and customers and suppliers that are from the same country.

Results: 195 records

FirstName	LastName	CustomerCountry	SupplierCountry	CompanyName
NULL	NULL	NULL	Australia	Pavlova, Ltd.

NULL	NULL	NULL	Australia	G'day, Mate
NULL	NULL	NULL	Japan	Tokyo Traders
NULL	NULL	NULL	Japan	Mayumi's
NULL	NULL	NULL	Netherlands	Zaanse Snoepfabriek
NULL	NULL	NULL	Singapore	Leka Trading
Patricio	Simpson	Argentina	NULL	NULL
Yvonne	Moncada	Argentina	NULL	NULL
Sergio	Gutiérrez	Argentina	NULL	NULL
Georg	Pipps	Austria	NULL	NULL
Roland	Mendel	Austria	NULL	NULL
Pascale	Cartrain	Belgium	NULL	NULL

Catherine	Dewey	Belgium	NULL	NULL
Bernardo	Batista	Brazil	Brazil	Refrescos Americanas LTDA
Lúcia	Carvalho	Brazil	Brazil	Refrescos Americanas LTDA
Janete	Limeira	Brazil	Brazil	Refrescos Americanas LTDA
Aria	Cruz	Brazil	Brazil	Refrescos Americanas LTDA
André	Fonseca	Brazil	Brazil	Refrescos Americanas LTDA
Mario	Pontes	Brazil	Brazil	Refrescos Americanas LTDA
Pedro	Afonso	Brazil	Brazil	Refrescos Americanas LTDA
Paula	Parente	Brazil	Brazil	Refrescos Americanas LTDA
Anabela	Domingues	Brazil	Brazil	Refrescos Americanas LTDA
Elizabeth	Lincoln	Canada	Canada	Ma Maison

Elizabeth	Lincoln	Canada	Canada	Forêts d'érables
Yoshi	Tannamuri	Canada	Canada	Ma Maison
Yoshi	Tannamuri	Canada	Canada	Forêts d'érables
Jean	Fresnière	Canada	Canada	Ma Maison

MySQL does not support FULL JOIN, you can use **UNION ALL** clause to combine these two JOINS as shown below.

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
        FROM CUSTOMERS
        LEFT JOIN ORDERS
        ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION ALL
        SELECT  ID, NAME, AMOUNT, DATE
        FROM CUSTOMERS
        RIGHT JOIN ORDERS
```

```
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
```

SQL - CARTESIAN or CROSS JOINS

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

Syntax

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

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

OR

```
SELECT * FROM [TABLE1] CROSS JOIN [TABLE2]
```

IF NO WHERE CLAUSE IN TABLE1, TABLE2 THEN CROSS JOIN ELSE INNER JOIN IF WHERE CLAUSE ON ID

BEST EXAMPLE IPL MATCHES

Example

Consider the following two tables.

Table 1 – CUSTOMERS table is as follows.

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

Table 2: ORDERS Table is as follows –

|--|--|--|--|--|

OID	DATE	CUSTOMER_ID	AMOUNT
102	2009-10-08 00:00:00	3	3000
100	2009-10-08 00:00:00	3	1500
101	2009-11-20 00:00:00	2	1560
103	2008-05-20 00:00:00	4	2060

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

```
SQL> SELECT ID, NAME, AMOUNT, DATE
      FROM CUSTOMERS, ORDERS;
```

This would produce the following result –

ID	NAME	AMOUNT	DATE
1	Ramesh	3000	2009-10-08 00:00:00
1	Ramesh	1500	2009-10-08 00:00:00
1	Ramesh	1560	2009-11-20 00:00:00
1	Ramesh	2060	2008-05-20 00:00:00
2	Khilan	3000	2009-10-08 00:00:00

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

SQL Self JOIN

- A self JOIN occurs when a table takes a 'selfie'.
- A self JOIN is a regular join but the table is joined with itself.
- This can be useful when modeling hierarchies.
- They are also useful for comparisons within a table.

SQL Self JOIN Examples

Problem: Match customers that are from the same city and country

```
1.      SELECT B.FirstName AS FirstName1, B.LastName AS LastName1,
2.          A.FirstName AS FirstName2, A.LastName AS LastName2,
3.          B.City, B.Country
4.      FROM Customer A, Customer B
5.      WHERE A.Id <> B.Id
```

```

6.      AND A.City = B.City
7.      AND A.Country = B.Country
8.      ORDER BY A.Country

```

A and B are aliases for the same Customer table.

Results: 88 records

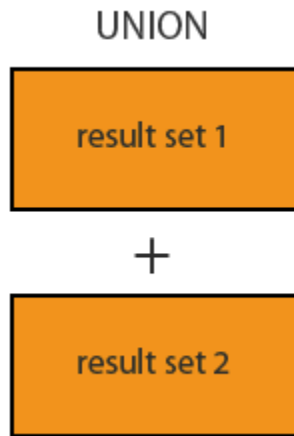
FirstName1	LastName1	FirstName2	LastName2	City	Country
Patricio	Simpson	Yvonne	Moncada	Buenos Aires	Argentina
Patricio	Simpson	Sergio	Gutiérrez	Buenos Aires	Argentina
Yvonne	Moncada	Patricio	Simpson	Buenos Aires	Argentina
Yvonne	Moncada	Sergio	Gutiérrez	Buenos Aires	Argentina
Sergio	Gutiérrez	Patricio	Simpson	Buenos Aires	Argentina
Sergio	Gutiérrez	Yvonne	Moncada	Buenos Aires	Argentina

Anabela	Domingues	Lúcia	Carvalho	Sao Paulo	Brazil
Anabela	Domingues	Aria	Cruz	Sao Paulo	Brazil
Anabela	Domingues	Pedro	Afonso	Sao Paulo	Brazil
Bernardo	Batista	Janete	Limeira	Rio de Janeiro	Brazil
Bernardo	Batista	Mario	Pontes	Rio de Janeiro	Brazil
Lúcia	Carvalho	Anabela	Domingues	Sao Paulo	Brazil
Lúcia	Carvalho	Aria	Cruz	Sao Paulo	Brazil
Lúcia	Carvalho	Pedro	Afonso	Sao Paulo	Brazil
Janete	Limeira	Bernardo	Batista	Rio de Janeiro	Brazil
Janete	Limeira	Mario	Pontes	Rio de Janeiro	Brazil
Aria	Cruz	Anabela	Domingues	Sao Paulo	Brazil

Aria	Cruz	Lúcia	Carvalho	Sao Paulo	Brazil
Aria	Cruz	Pedro	Afonso	Sao Paulo	Brazil
Mario	Pontes	Bernardo	Batista	Rio de Janeiro	Brazil
Mario	Pontes	Janete	Limeira	Rio de Janeiro	Brazil
Pedro	Afonso	Anabela	Domingues	Sao Paulo	Brazil
Pedro	Afonso	Lúcia	Carvalho	Sao Paulo	Brazil
Pedro	Afonso	Aria	Cruz	Sao Paulo	Brazil
Dominique	Perrier	Marie	Bertrand	Paris	France
Marie	Bertrand	Dominique	Perrier	Paris	France
Janine	Labrune	Carine	Schmitt	Nantes	France
Carine	Schmitt	Janine	Labrune	Nantes	France

SQL UNION Clause

- UNION combines the result sets of two queries.
- Column data types in the two queries must match.
- UNION combines by column position rather than column name.



The SQL UNION syntax

The general syntax is:

```
1.      SELECT column-names
```



```
2.      FROM table-name
3.      UNION
4.      SELECT column-names
5.      FROM table-name
```

SUPPLIER
Id
CompanyName
ContactName
City
Country
Phone
Fax
CUSTOMER
Id
FirstName

LastName
City
Country
Phone

SQL UNION Examples

Problem: List all contacts, i.e., suppliers and customers.

```
1.      SELECT 'Customer' As Type,  
2.          FirstName + ' ' + LastName AS ContactName,  
3.          City, Country, Phone  
4.      FROM Customer  
5.  UNION  
6.      SELECT 'Supplier',  
7.          ContactName, City, Country, Phone  
8.      FROM Supplier
```

This is a simple example in which the table alias would be useful

Results:

Type	ContactName	City	Country	Phone
Customer	Alejandra Camino	Madrid	Spain	(91) 745 6200
Customer	Alexander Feuer	Leipzig	Germany	0342-023176
Customer	Ana Trujillo	México D.F.	Mexico	(5) 555-4729
Customer	Anabela Domingues	Sao Paulo	Brazil	(11) 555-2167
...				
Supplier	Anne Heikkonen	Lappeenranta	Finland	(953) 10956
Supplier	Antonio del Valle Saavedra	Oviedo	Spain	(98) 598 76 54
Supplier	Beate Vileid	Sandvika	Norway	(0)2-953010

Supplier	Carlos Diaz	Sao Paulo	Brazil	(11) 555 4640
Supplier	Chandra Leka	Singapore	Singapore	555-8787
Supplier	Chantal Goulet	Ste-Hyacinthe	Canada	(514) 555-2955
Supplier	Charlotte Cooper	London	UK	(171) 555-2222

SQL Subqueries

- A subquery is a SQL query within a query.
- Subqueries are nested queries that provide data to the enclosing query.
- Subqueries can return individual values or a list of records
- Subqueries must be enclosed with parenthesis

The SQL subquery syntax

There is no general syntax; subqueries are regular queries placed inside parenthesis.
 Subqueries can be used in different ways and at different locations inside a query:
 Here is an subquery with the IN operator

```

1.      SELECT column-names
2.          FROM table-name1
3.      WHERE value IN (SELECT column-name
4.                      FROM table-name2
5.                      WHERE condition)

```

Subqueries can also assign column values for each record:

```

1.      SELECT column1 = (SELECT column-name FROM table-name WHERE con
2.          dition),
3.          column-names
4.      FROM table-name
5.      WEHRE condition

```

ORDERITEM

Id

OrderId
ProductId
UnitPrice
Quantity
PRODUCT
Id
ProductName
SupplierId
UnitPrice
Package
IsDiscontinued

SQL Subquery Examples

Problem: List products with order quantities greater than 100.

```
1.      SELECT ProductName
2.      FROM Product
3.      WHERE Id IN (SELECT ProductId
4.                  FROM OrderItem
5.                  WHERE Quantity > 100)
```

Results: 12 records

PoductName

Guaraná Fantástica

Schoggi Schokolade

Chartreuse verte

Jack's New England Clam Chowder

Rogede sild

Manjimup Dried Apples

Perth Pasties

...

CUSTOMER

Id

FirstName

LastName

City

Country

Phone

ORDER

Id

OrderDate
OrderNumber
CustomerId
TotalAmount

SQL Subquery Examples

Problem: List all customers with their total number of orders

```
1.      SELECT FirstName, LastName,
2.          OrderCount = (SELECT COUNT(O.Id) FROM [Order] O WHERE O
   .CustomerId = C.Id)
3.      FROM Customer C
```

This is a **correlated subquery** because the subquery references the enclosing query (i.e. the C.Id in the WHERE clause).

Results: 91 records

FirstName	LastName	OrderCount
Maria	Anders	6
Ana	Trujillo	4
Antonio	Moreno	7
Thomas	Hardy	13
Christina	Berglund	18
Hanna	Moos	7
Frédérique	Citeaux	11
Martín	Sommer	3

...

SQL **WHERE EXISTS** Statement

- WHERE EXISTS tests for the existence of any records in a subquery.
- EXISTS returns true if the subquery returns one or more records.
- EXISTS is commonly used with correlated subqueries.

The SQL EXISTS syntax

The general syntax is:

```
1.      SELECT column-names
2.      FROM table-name
3.      WHERE EXISTS
4.      (SELECT column-name
5.      FROM table-name
6.      WHERE condition)
```

SUPPLIER
Id
CompanyName
ContactName
City
Country
Phone
Fax
PRODUCT
Id
ProductName
SupplierId
UnitPrice
Package
IsDiscontinued

SQL EXISTS Example

Problem: Find suppliers with products over \$100.

```
1.      SELECT CompanyName
2.      FROM Supplier A
3.      WHERE EXISTS
4.          (SELECT ProductName
5.             FROM Product B
6.             WHERE B.SupplierId = A.Supplier.Id
7.             AND UnitPrice > 100)
```

This is a **correlated subquery** because the subquery references the enclosing query (with Supplier.Id).

Results: 2 records

						PROD ID		SUPPLIER ID		
S1		1 S1				1 P1		1	50	
						2 P2		1	120	BREAK
						3 P3		1	145	
		2 S2								
						1 P1		2	67	
						3 P3		2	84	
S3										
		3 S3				1 P1		3	122	
						2 P2		3	120	
						3 P3		3	84	

CompanyName

Plutzer Lebensmittelgroßmärkte AG

Aux joyeux ecclésiastiques

[illegible]

Views

A view can be simply thought of as a SQL query stored permanently on the server. Whatever indices the query optimizes to will be used. In that sense, there is no difference between the SQL query or a view. It does not affect performance any more negatively than the actual SQL query. If anything, since it is stored on the server, and does not need to be evaluated at run time, it is actually faster.

It does afford you these additional advantages

- reusability
- a single source for optimization
-

Data is auto refreshed when view query is fired.

Creating MySQL view examples

Creating simple views

Let's take a look at the `orderDetails` table. We can create a view that represents total sales per order.

```
1 CREATE VIEW SalePerOrder AS
```

```
2  SELECT
```



```
3    orderNumber, SUM(quantityOrdered * priceEach) total
4  FROM
5    orderDetails
6  GROUP BY orderNumber
7  ORDER BY total DESC;
```

If you use the `SHOW TABLE` command to view all tables in the `classicmodels` database, we also see the `SalesPerOrder` view is showing up in the list.

```
1 SHOW TABLES;
```

	Tables_in_classicmodels
▶	customers
	employees
	offices
	orderdetails
	orders
	payments
	productlines
	products
	saleperorder

This is because the views and tables share the same namespace. To know which object is view or table, you use the `SHOW FULL TABLE` command as follows:

	Tables_in_classicmodels	Table_type
▶	customers	BASE TABLE
	employees	BASE TABLE
	offices	BASE TABLE
	orderdetails	BASE TABLE
	orders	BASE TABLE
	payments	BASE TABLE
	productlines	BASE TABLE
	products	BASE TABLE
	saleperorder	VIEW

The `table_type` column in the result set specifies which object is view and which object is a table (base table).

If we want to query total sales for each sales order, you just need to execute a simple `SELECT` statement against the `salePerOrder` view as follows:

1 `SELECT`

2 `*`

3 `FROM`

4 `salePerOrder;`

	orderNumber	total
▶	10165	67392.84
	10287	61402
	10310	61234.66
	10212	59830.54
	10207	59265.14
	10127	58841.35
	10204	58793.53
	10126	57131.92

Creating a view based on another view

MySQL allows you to create a view based on another view. For example, you can create a view called big sales order based on the `SalesPerOrder` view to show every sales order whose total is greater than `60,000` as follows:

```
1 CREATE VIEW BigSalesOrder AS
2   SELECT
3     orderNumber, ROUND(total,2) as total
4   FROM
5     saleperorder
```

```
6 WHERE
7     total > 60000;
```

Now, we can query the data from the `BigSalesOrder` view as follows:

```
1 SELECT
2     orderNumber, total
3 FROM
4     BigSalesOrder;
```

	orderNumber	total
▶	10165	67392.85
	10287	61402.00
	10310	61234.67

Creating views with join

The following is an example of creating a view with [INNER JOIN](#) . The view contains the *order number*, *customer name*, and *total sales* per order.

```
1 CREATE VIEW customerOrders AS
```

```
2  SELECT
3      d.orderNumber,
4      customerName,
5      SUM(quantityOrdered * priceEach) total
6  FROM
7      orderDetails d
8      INNER JOIN
9      orders o ON o.orderNumber = d.orderNumber
10     INNER JOIN
11     customers c ON c.customerNumber = o.customerNumber
12  GROUP BY d.orderNumber
13  ORDER BY total DESC;
```

To query data from the `customerOrders` view, you use the following query:

```
1 SELECT
```

2 *

3 FROM

4 customerOrders;

	orderNumber	customerName	total
▶	10165	Dragon Souvenirs, Ltd.	67392.84
	10287	Vida Sport, Ltd	61402
	10310	Toms Spezialitäten, Ltd	61234.66
	10212	Euro+ Shopping Channel	59830.54
	10207	Diecast Collectables	59265.14
	10127	Muscle Machine Inc	58841.35
	10204	Muscle Machine Inc	58793.53
	10126	Comida Auto Replicas, Ltd	57131.92

Creating views with subquery

The following illustrates how to create a view with a [subquery](#). The view contains products whose buy prices are higher than the average price of all products.

1 CREATE VIEW aboveAvgProducts AS

2 SELECT

3 productCode, productName, buyPrice

```
4  FROM
5    products
6  WHERE
7    buyPrice >
8  (SELECT
9    AVG(buyPrice)
10   FROM
11    products)
12 ORDER BY buyPrice DESC;
```

Querying data from the `aboveAvgProducts` is simple as follows:

```
1 SELECT
2   *
3 FROM
4   aboveAvgProducts;
```

	productCode	productName	buyPrice
►	S10_4962	1962 LanciaA Delta 16V	103.42
	S18_2238	1998 Chrysler Plymouth Prowler	101.51
	S10_1949	1952 Alpine Renault 1300	98.58
	S24_3856	1956 Porsche 356A Coupe	98.3
	S12_1108	2001 Ferrari Enzo	95.59
	S12_1099	1968 Ford Mustang	95.34
	S18_1984	1995 Honda Civic	93.89
	S18_4027	1970 Triumph Spitfire	91.92

SQL - Transactions

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

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

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

Properties of Transactions

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

- **Atomicity** – ensures that all operations within the work unit are completed successfully. Otherwise, the transaction is aborted at the point of failure and all the previous operations are rolled back to their former state.
- **Consistency** – ensures that the database properly changes states upon a successfully committed transaction.
- **Isolation** – enables transactions to operate independently of and transparent to each other.
- **Durability** – ensures that the result or effect of a committed transaction persists in case of a system failure.

Transaction Control

The following commands are used to control transactions.

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

Transactional Control Commands

Transactional control commands are only used with the **DML Commands** such as - INSERT, UPDATE and DELETE only. They cannot be used while creating tables or dropping them because these operations are automatically committed in the database.

The COMMIT Command

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

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database. The COMMIT command saves all the transactions to the database since the last COMMIT or ROLLBACK command.

The syntax for the COMMIT command is as follows.

```
COMMIT;
```

Example

Consider the CUSTOMERS table having the following records –

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

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

```
SQL> DELETE FROM CUSTOMERS
      WHERE AGE = 25;
SQL> COMMIT;
```

Thus, two rows from the table would be deleted and the SELECT statement would produce the following result.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
3	kaushik	23	Kota	2000.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

The ROLLBACK Command

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database. This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

The syntax for a ROLLBACK command is as follows –

```
ROLLBACK;
```

Example

Consider the CUSTOMERS table having the following records –

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

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

```
SQL> DELETE FROM CUSTOMERS
      WHERE AGE = 25;
SQL> ROLLBACK;
```

Thus, the delete operation would not impact the table and the SELECT statement would produce the following result.

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

The SAVEPOINT Command

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

The syntax for a SAVEPOINT command is as shown below.

```
SAVEPOINT SAVEPOINT_NAME;
```

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

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

```
ROLLBACK TO SAVEPOINT_NAME;
```

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

Example

Consider the CUSTOMERS table having the following records.

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

The following code block contains the series of operations.

```
SQL> SAVEPOINT SP1;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=1;
1 row deleted.
```



```
SQL> SAVEPOINT SP2;  
Savepoint created.  
SQL> DELETE FROM CUSTOMERS WHERE ID=2;  
1 row deleted.  
SQL> SAVEPOINT SP3;  
Savepoint created.  
SQL> DELETE FROM CUSTOMERS WHERE ID=3;  
1 row deleted.
```

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

```
SQL> ROLLBACK TO SP2;  
Rollback complete.
```

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

```
SQL> SELECT * FROM CUSTOMERS;
```

```

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

```

The RELEASE SAVEPOINT Command

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

The syntax for a RELEASE SAVEPOINT command is as follows.

```
RELEASE SAVEPOINT SAVEPOINT_NAME;
```

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

The SET TRANSACTION Command

The SET TRANSACTION command can be used to initiate a database transaction. This command is used to specify characteristics for the transaction that follows. For example, you can specify a transaction to be read only or read write.

The syntax for a SET TRANSACTION command is as follows.

```
SET TRANSACTION [ READ WRITE | READ ONLY ] ;
```

How to stop autocommit false for testing

```
SET SESSION autocommit = 0;
```

As of MySQL Workbench 6.0.0, you can set the "Leave autocommit mode enabled by default" preference. Set it under Preferences --> SQL Queries --> General.

SQL Injection

- SQL Injection is a code injection technique.
- It is the placement of malicious code in SQL strings.
- SQL Injection is one of the most common web hacking techniques.
- These attacks only work with apps that internally use SQL.

SQL Injection is an attack that poisons dynamic SQL statements to comment out certain parts of the statement or appending a condition that will always be true. It takes advantage of the design flaws in poorly designed web applications to exploit SQL statements to execute malicious SQL code.

SQL Injection Based on 1=1 is Always True

Look at the example above again. The original purpose of the code was to create an SQL statement to select a user, with a given user id.

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

UserId: 105 OR 1=1

Then, the SQL statement will look like this:

```
SELECT * FROM Users WHERE UserId = 105 OR 1=1;
```

The SQL above is valid and will return ALL rows from the "Users" table, since **OR 1=1** is always TRUE.

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

The SQL statement above is much the same as this:

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

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

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

Here is an example of a user login on a web site:

Username:

John Doe

Password:

myPass

Example

```
uName = getQueryString("username");  
uPass = getQueryString("userpassword");
```

```
sql = 'SELECT * FROM Users WHERE Name =' + uName + ' ' AND Pass =' +  
uPass + ' ';
```

Result

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

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

User Name:

Password:

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

Result

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

The SQL above is valid and will return all rows from the "Users" table, since **OR** ""="" is always TRUE.

SQL Injection Based on Batched SQL Statements

Most databases support batched SQL statement.

A batch of SQL statements is a group of two or more SQL statements, separated by semicolons.

The SQL statement below will return all rows from the "Users" table, then delete the "Suppliers" table.

Example

```
SELECT * FROM Users; DROP TABLE Suppliers
```

Look at the following example:

Example

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

And the following input:

User id: 105; DROP TABLE Suppliers

The valid SQL statement would look like this:

Result

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

Explaining my.ini or my.cnf

MySQL Server Instance Configuration File

Add this to /etc/my.cnf (Linux) or my.ini (Windows)

```
[mysqld]
```

```
autocommit=0
```

or

The maximum amount of concurrent sessions the MySQL server will allow. One of these connections will be reserved for a user with SUPER privileges to allow the administrator to login even if the connection limit has been reached.

```
max_connections=100
```

changed to

```
max_connections = 1000
```

then restart mysql

MySQL show status - Open database connections

MySQL SERVER

- It is multithreaded.

- It (usually) acts as a TCP/IP server, accepting connections.
- Each connection gets a dedicated thread.

You can show MySQL open database connections (and other MySQL database parameters) using the MySQL `show status` command, like this:

```
mysql> show status like '%onn%';
```

Variable_name	Value
Aborted_connects	0
Connections	8
Max_used_connections	4
Ssl_client_connects	0
Ssl_connect_renegotiates	0

```

| Ssl_finished_connects      | 0      |
| Threads_connected          | 4      |
+-----+-----+
7 rows in set (0.00 sec)

```

All those rows and values that are printed out correspond to MySQL variables that you can look at. Notice that I use `like 'Conn%'` in the first example to show variables that look like "Connection", then got a little wiser in my second MySQL show status query.

MySQL show processlist

Here's what my MySQL processlist looks like when I had my Java application actively running under Tomcat:

```

mysql> show processlist;

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

```

Id	User	Host	db	Command	Time	State
3	root	localhost	webapp	Query	0	NULL
5	root	localhost:61704	webapp	Sleep	208	
6	root	localhost:61705	webapp	Sleep	208	
7	root	localhost:61706	webapp	Sleep	208	

4 rows in set (0.00 sec)

How To Measure MySQL Query Performance with mysqlslap

MySQL comes with a handy little diagnostic tool called **mysqlslap** that's been around since version 5.1.4. It's a benchmarking tool that can help DBAs and developers load test their database servers.

mysqlslap can emulate a large number of client connections hitting the database server at the same time. The load testing parameters are fully configurable and the results from different test runs can be used to fine-tune database design or hardware resources.

In this tutorial we will learn how to use mysqlslap to load test a MySQL database with some basic queries and see how benchmarking can help us fine-tune those queries. After some basic demonstrations, we will run through a fairly realistic test scenario where we create a copy of an existing database for testing, glean queries from a log, and run the test from a script.

Using mysqlslap

We can now start using mysqlslap. mysqlslap can be invoked from a regular shell prompt so there's no need to explicitly log in to MySQL. For this tutorial, though, we will open another terminal connection to our Linux server and start a new MySQL session from there with the **sysadmin** user we created before, so we can check and update a few things in MySQL more easily. So, we'll have one prompt open with our sudo user, and one prompt logged into MySQL.

Before we get into specific commands for testing, you may want to take a look at this list of the most useful mysqlslap options. This can help you design your own mysqlslap commands later.

Option	What it means
--user	MySQL username to connect to the database server
--password	Password for the user account. It's best to leave it blank in command line
--host	MySQL database server name
--port	Port number for connecting to MySQL if the default is not used

Option	What it means
--concurrency	The number of simultaneous client connections mysqlslap will emulate
--iterations	The number of times the test query will be run
--create-schema	The database where the query will be run
--query	The query to execute. This can either be a SQL query string or a path to a SQL script file
--create	The query to create a table. Again, this can be a query string or a path to a SQL file
--delimiter	The delimiter used to separate multiple SQL statements
--engine	The MySQL database engine to use (e.g., InnoDB)
--auto-generate-sql	Lets MySQL perform load testing with its own auto-generated SQL command

```
sudo mysqlslap --user=sysadmin --password --host=localhost --
concurrency=20 --iterations=10 --create-schema=employees --
query="SELECT * FROM employees;SELECT * FROM titles;SELECT * FROM
```

```
dept_emp;SELECT * FROM dept_manager;SELECT * FROM departments;" --  
delimiter=";" --verbose
```

Benchmark

```
    Average number of seconds to run all queries: 23.800  
seconds  
    Minimum number of seconds to run all queries: 22.751  
seconds  
    Maximum number of seconds to run all queries: 26.788  
seconds  
    Number of clients running queries: 20  
    Average number of queries per client: 5
```

or

```
C:\Program Files\MySQL\MySQL Server 5.6\bin>mysqlslap --user=root --password --delimiter=";" --  
create-schema=eexam --query="SELECT * FROM BRANCH LIMIT 2" --concurrency=10 --iterations=1
```

Enter password: *****

Benchmark

```
    Average number of seconds to run all queries: 0.063 seconds
```

Minimum number of seconds to run all queries: 0.063 seconds

Maximum number of seconds to run all queries: 0.063 seconds

Number of clients running queries: 10

Average number of queries per client: 1

MySQL Stored Procedure

A procedure (often called a stored procedure) is a subroutine like a subprogram in a regular computing language, stored in database. A procedure has a name, a parameter list, and SQL statement(s). All most all relational database system supports stored procedure, MySQL 5 introduce stored procedure. In the following sections we have discussed MySQL procedure in details and used MySQL 5.6 under Windows 7. MySQL 5.6 supports "routines" and there are two kinds of routines : stored procedures which you call, or functions whose return values you use in other SQL statements the same way that you use pre-installed MySQL functions like pi(). The major difference is that UDFs can be used like any other expression within SQL statements, whereas stored procedures must be invoked using the CALL statement.

Short example:

```
DELIMITER $$
```

```
CREATE PROCEDURE mybranch()
```

```
BEGIN
```

```
SELECT * FROM BRANCH;
```

```
END;
```

```
$$
```

```
call eexam.mybranch();
```

Why Stored Procedures?

- Stored procedures are fast. MySQL server takes some advantage of caching, just as prepared statements do. The main speed gain comes from reduction of network traffic. If you have a repetitive task that requires checking, looping, multiple statements, and no user interaction, do it with a single call to a procedure that's stored on the server.
- Stored procedures are portable. When you write your stored procedure in SQL, you know that it will run on every platform that MySQL runs on, without obliging you to install an additional runtime-environment package, or set permissions for program execution in the operating system, or deploy different packages if you have different computer types.

That's the advantage of writing in SQL rather than in an external language like Java or C or PHP.

- Stored procedures are always available as 'source code' in the database itself. And it makes sense to link the data with the processes that operate on the data.
- Stored procedures are migratory! MySQL adheres fairly closely to the SQL:2003 standard. Others (DB2, Mimer) also adhere.

Create Procedure

Following statements create a stored procedure. By default, a procedure is associated with the default database (currently used database). To associate the procedure with a given database, specify the name as database_name.stored_procedure_name when you create it. Here is the complete syntax :

Syntax :

```
CREATE [DEFINER = { user | CURRENT_USER }]
```

```
PROCEDURE sp_name ([proc_parameter[,...]])
```

```
[characteristic ...] routine_body
```

```
proc_parameter: [ IN | OUT | INOUT ] param_name type
```

```
type:
```

Any valid MySQL data type

characteristic:

COMMENT 'string'

| LANGUAGE SQL

| [NOT] DETERMINISTIC

| { CONTAINS SQL | NO SQL | READS SQL DATA

| MODIFIES SQL DATA }

| SQL SECURITY { DEFINER | INVOKER }

routine_body:

Valid SQL routine statement

Before create a procedure we need some information which are described in this section :

Check the MySQL version :

Following command displays the version of MySQL :

```
mysql>SELECT VERSION();
```

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

```
| VERSION() |
```

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

```
| 5.6.12    |
```

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

```
1 row in set (0.00 sec)
```

Check the privileges of the current user :

CREATE PROCEDURE and CREATE FUNCTION require the CREATE ROUTINE privilege. They might also require the SUPER privilege, depending on the DEFINER value, as described later in this section. If binary logging is enabled, CREATE FUNCTION might require the SUPER privilege. By default, MySQL automatically grants the ALTER ROUTINE and EXECUTE privileges to the routine creator. This behavior can be changed by disabling the `automatic_sp_privileges` system variable.

```
mysql> SHOW PRIVILEGES;
```

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

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

```
| Privilege          | Context          | Comment
```

```
|
```

+-----+-----+-----+		
-----+		
Alter	Tables	To alter the table
Alter routine	Functions,Procedures	To alter or drop s
tored functions/procedures		
Create	Databases,Tables,Indexes	To create new data
bases and tables		
Create routine	Databases	To use CREATE FUNC
TION/PROCEDURE		
Create temporary	Databases	To use CREATE TEMP
ORARY TABLE		
tables		
Create view	Tables	To create new view
s		
Create user	Server Admin	To create new user
s		
Delete	Tables	To delete existing
rows		
Drop	Databases,Tables	To drop databases,
tables, and views		
Event	Server Admin	To create, alter,
drop and execute events		

Execute routines	Functions, Procedures	To execute stored
File files on the server	File access on server	To read and write
Grant option	Databases, Tables,	To give to other u
sers those privileges you possess	Functions, Procedures	
Index indexes	Tables	To create or drop
Insert o tables	Tables	To insert data int
Lock tables (together with SELECT privilege)	Databases	To use LOCK TABLES
Process text of currently executing queries	Server Admin	To view the plain
Proxy possible	Server Admin	To make proxy user
References on tables	Databases, Tables	To have references
Reload sh tables, logs and privileges	Server Admin	To reload or refre
Replication lave or master servers are	Server Admin	To ask where the s

client		
Replication events from the master	Server Admin	To read binary log
slave		
Select from table	Tables	To retrieve rows f
Show databases with SHOW DATABASES	Server Admin	To see all databas
Show view SHOW CREATE VIEW	Tables	To see views with
Shutdown server	Server Admin	To shut down the s
Super, SET GLOBAL, CHANGE MASTER, etc.	Server Admin	To use KILL thread
Trigger	Tables	To use triggers
Create op tablespaces	Server Admin	To create/alter/dr
tablespace		
Update rows	Tables	To update existing

Usage	Server Admin	No privileges - al
low connect only		

31 rows in set (0.00 sec)

Select a database :

Before creates a procedure we must select a database. Let see the list of databases and choose one of them.

```
mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| information_schema |
| hr |
| mysql |
| performance_schema |
| sakila |
| test |
| world |
+-----+
```

```
7 rows in set (0.06 sec))
```

Now select the database 'hr' and list the tables :

```
mysql> USE hr;
```

Database changed

```
mysql> SHOW TABLES;
```

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

```
| Tables_in_hr |
```

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

```
| alluser      |
```

```
| departments  |
```

```
| emp_details  |
```

```
| job_history  |
```

```
| jobs         |
```

```
| locations    |
```

```
| regions      |
```

```
| user         |
```

```
| user_details |
```

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

```
9 rows in set (0.00 sec)
```

Pick a Delimiter

The delimiter is the character or string of characters which is used to complete an SQL statement. By default we use semicolon (;) as a delimiter. But this causes problem in stored procedure because a procedure can have many statements, and everyone must end with a semicolon. So for your delimiter, pick a string which is rarely occur within statement or within procedure. Here we have used double dollar sign i.e. \$\$. You can use whatever you want. To resume using ";" as a delimiter later, say "DELIMITER ; \$\$". See here how to change the delimiter :

```
mysql> DELIMITER $$ ;
```

Now the default DELIMITER is "\$\$". Let execute a simple SQL command :

```
mysql> SELECT * FROM user $$
```

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

```
| userid   | password | name   |
```

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

```
| scott123 | 123@sco  | Scott |
```

```
| ferp6734 | dloeu@&3 | Palash |
```

```
| diana094 | ku$j@23  | Diana |
```

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

```
3 rows in set (0.00 sec)
```

Now execute the following command to resume ";" as a delimiter :

```
mysql> DELIMITER ; $$
```

Example : MySQL Procedure

Here we have created a simple procedure called job_data, when we will execute the procedure it will display all the data from "jobs" tables.

```
mysql> DELIMITER $$ ;mysql> CREATE PROCEDURE job_data()  
-> SELECT * FROM JOBS; $$  
Query OK, 0 rows affected (0.00 sec)
```

Explanation :

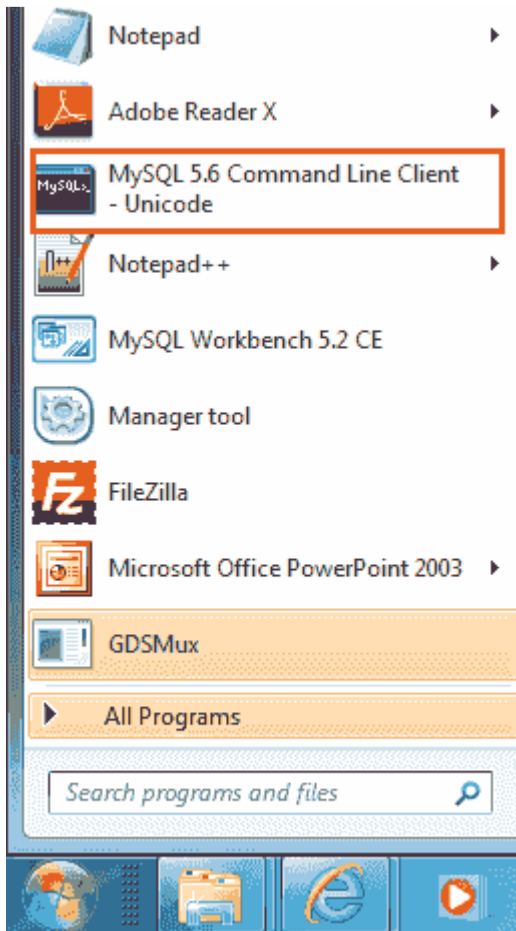
- CREATE PROCEDURE command creates the stored procedure.
- Next part is the procedure name. Here the procedure name is "job_data".
- Procedure names are not case sensitive, so job_data and JOB_DATA are same.
- You cannot use two procedures with the same name in the same database.
- You can use qualified names of the form "database-name.procedure-name", for example "hr.job_data".
- Procedure names can be delimited. If the name is delimited, it can contain spaces.
- The maximum name length is 64 characters.
- Avoid using names of built-in MySQL functions.
- The last part of "CREATE PROCEDURE" is a pair of parentheses. "()" holds the parameter(s) list as there are no parameters in this procedure, the parameter list is empty.
- Next part is SELECT * FROM JOBS; \$\$ which is the last statement of the procedure body. Here the semicolon (;) is optional as \$\$ is a real statement-ender.

Tools to create MySQL Procedure

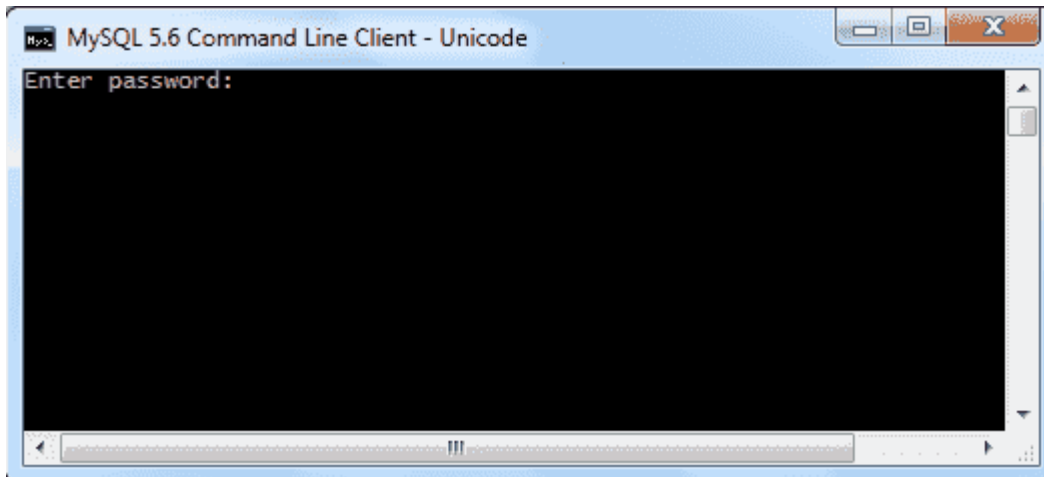
You can write a procedure in MySQL command line tool or you can use [MySQL workbench](#) which is an excellent front-end tool (here we have used version 5.3 CE).

MySQL command line tool : -

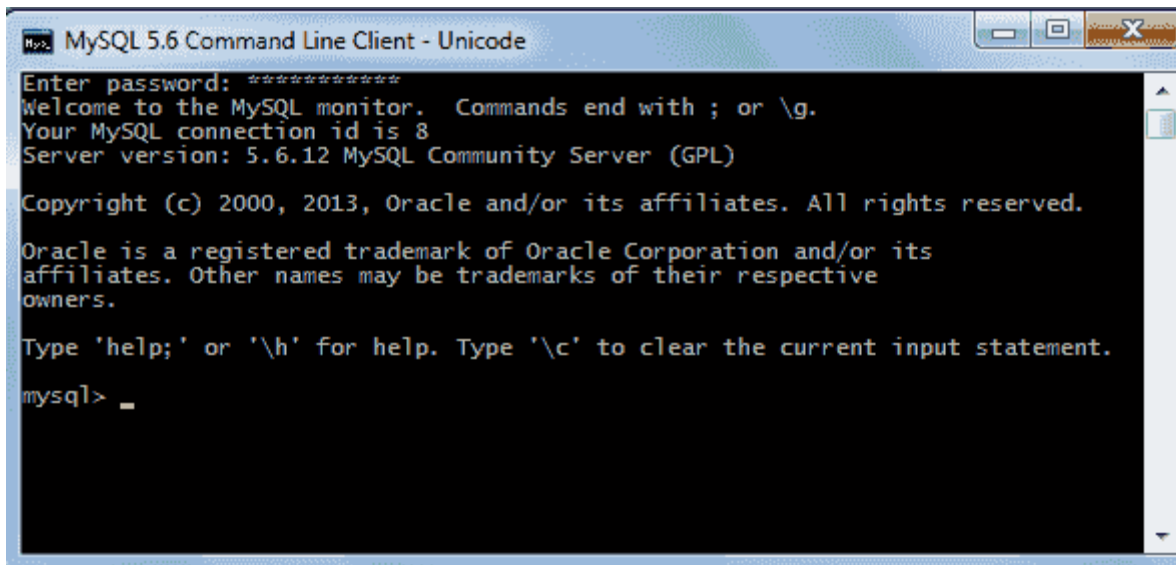
Select MySQL command Client from Start menu :



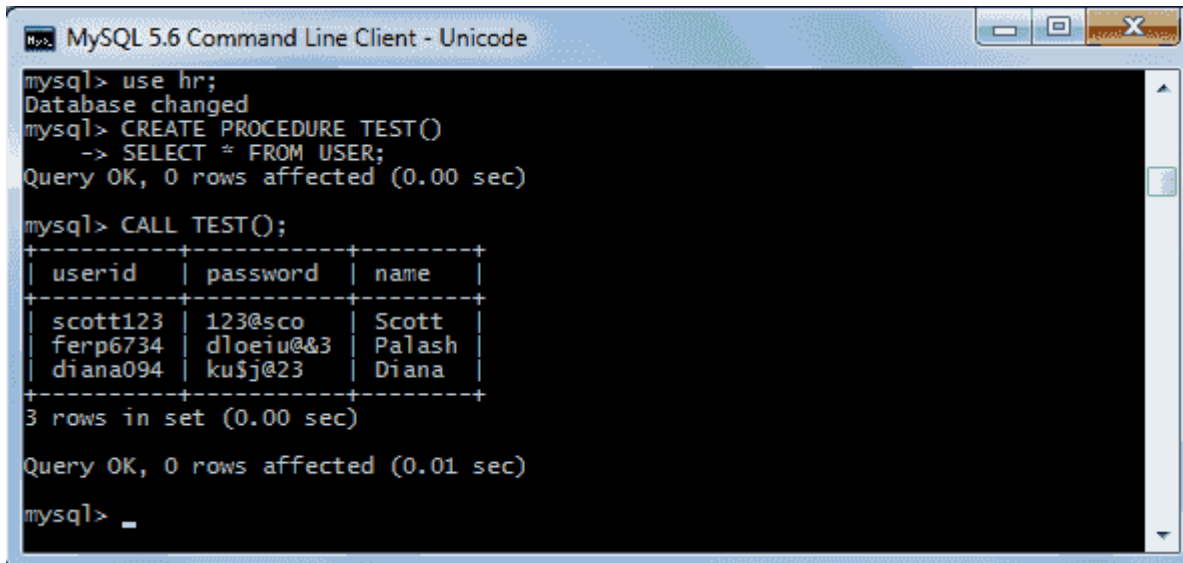
Selecting MySQL command prompt following screen will come :



After a successful login you can access the MySQL command prompt :



Now you write and run your own procedure, see the following example :



```
mysql> use hr;
Database changed
mysql> CREATE PROCEDURE TEST()
-> SELECT * FROM USER;
Query OK, 0 rows affected (0.00 sec)

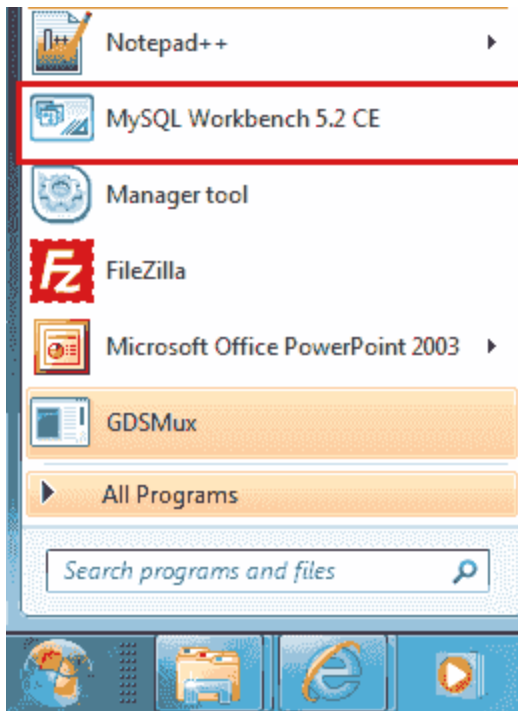
mysql> CALL TEST();
+-----+-----+-----+
| userid | password | name |
+-----+-----+-----+
| scott123 | 123@sco | Scott |
| ferp6734 | dloeuu@&3 | Palash |
| diana094 | ku$j@23 | Diana |
+-----+-----+-----+
3 rows in set (0.00 sec)

Query OK, 0 rows affected (0.01 sec)

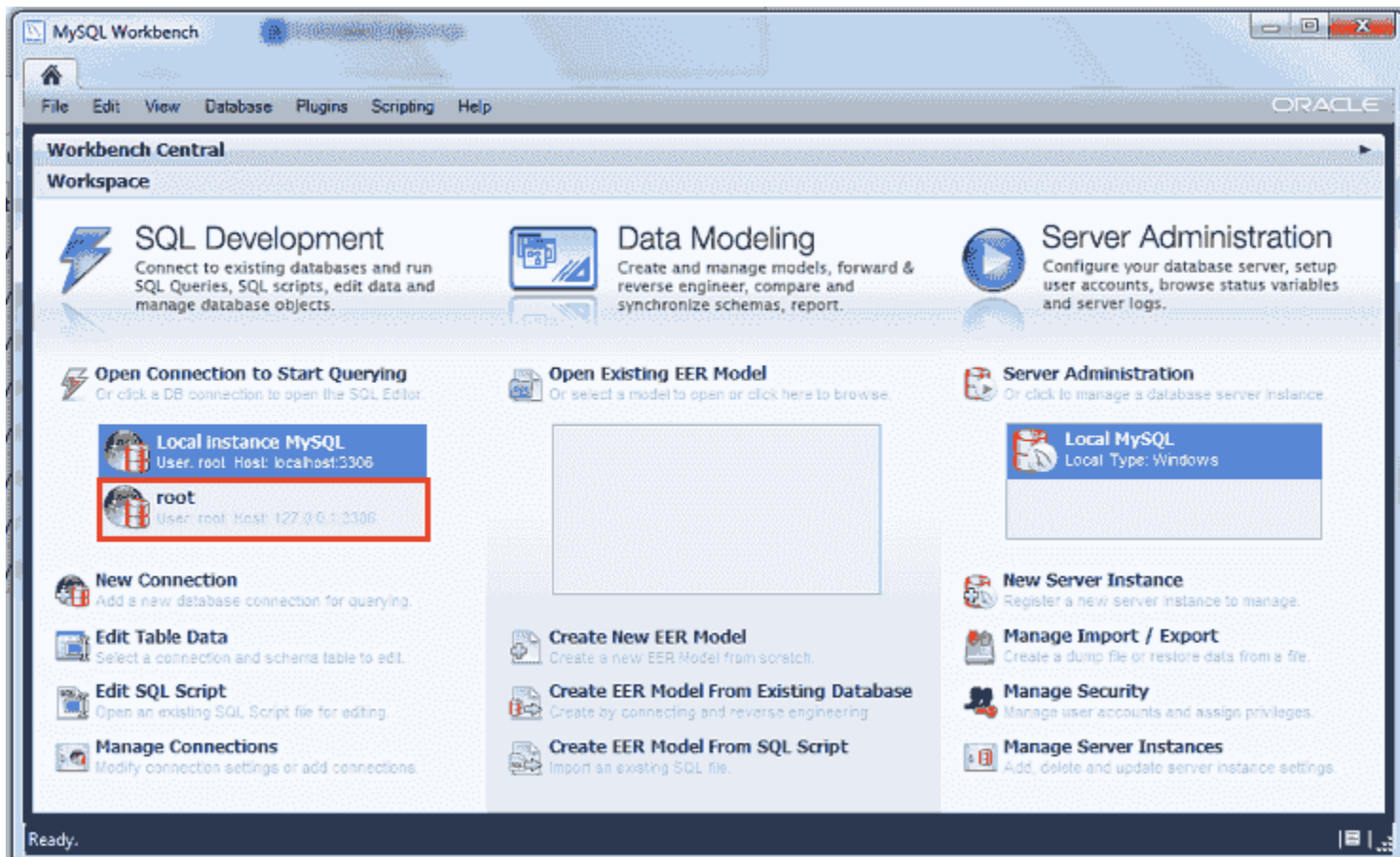
mysql> _
```

MySQL workbench (5.3 CE) : -

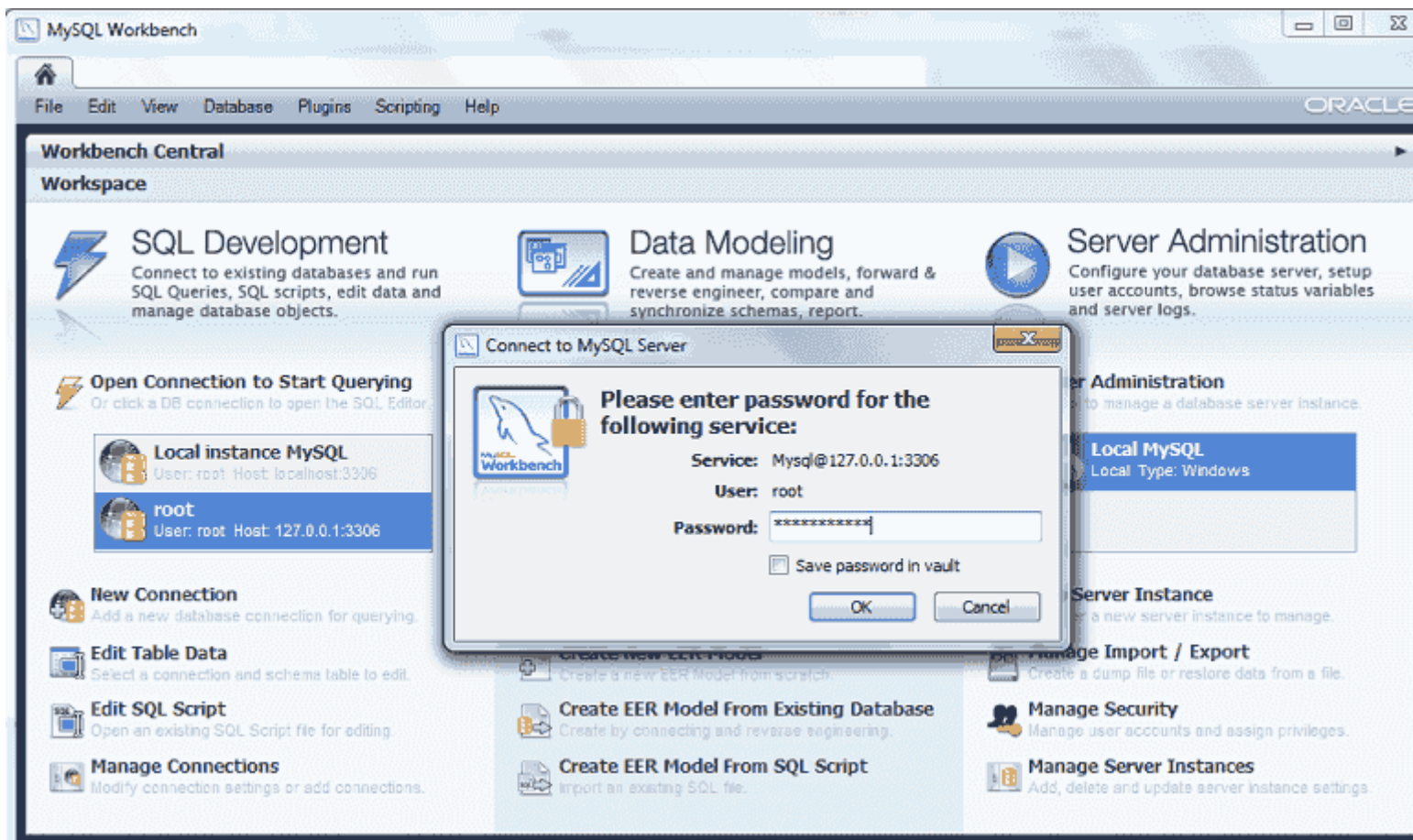
Select MySQL workbench from Start menu :



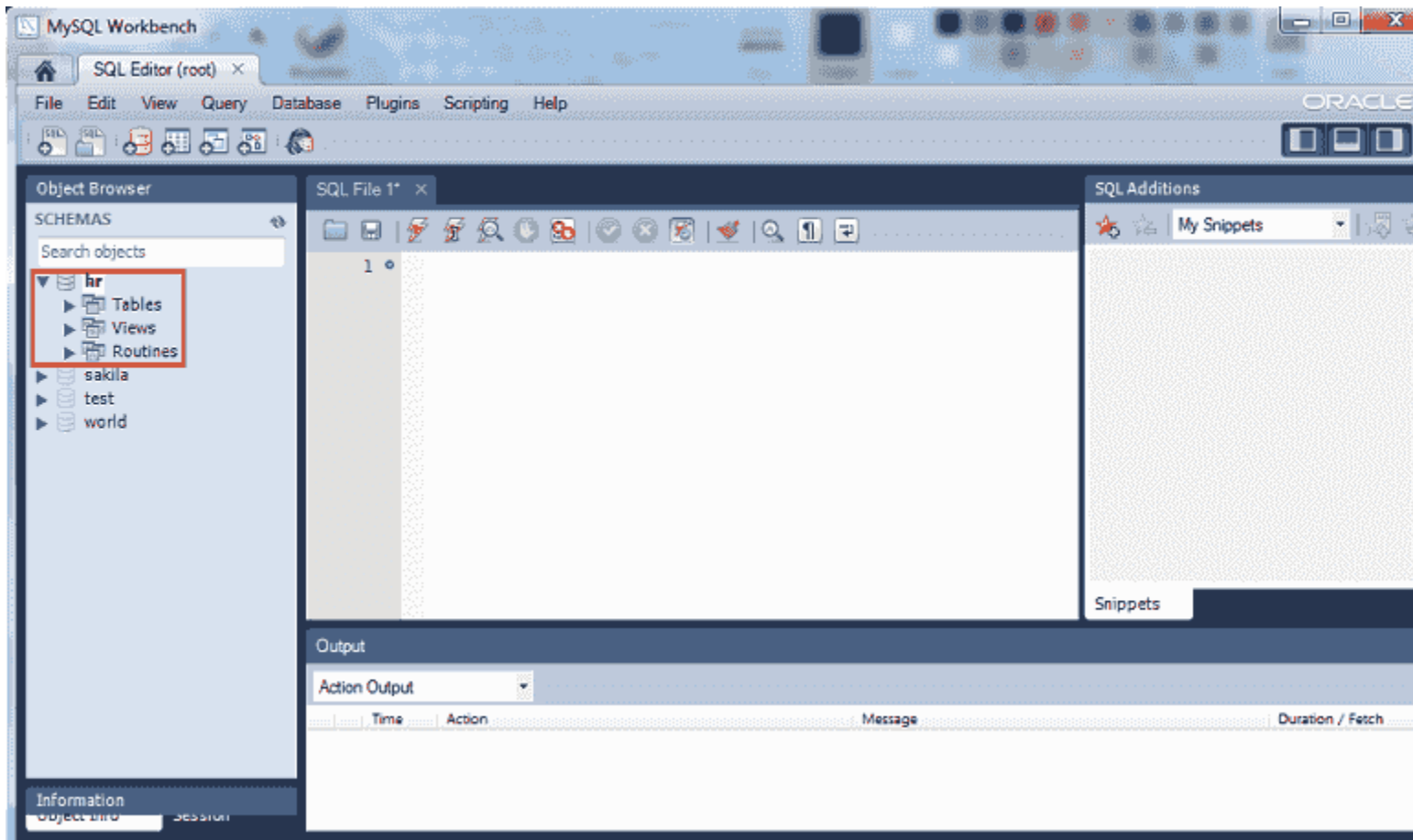
After selecting MySQL workbench following login screen will come :



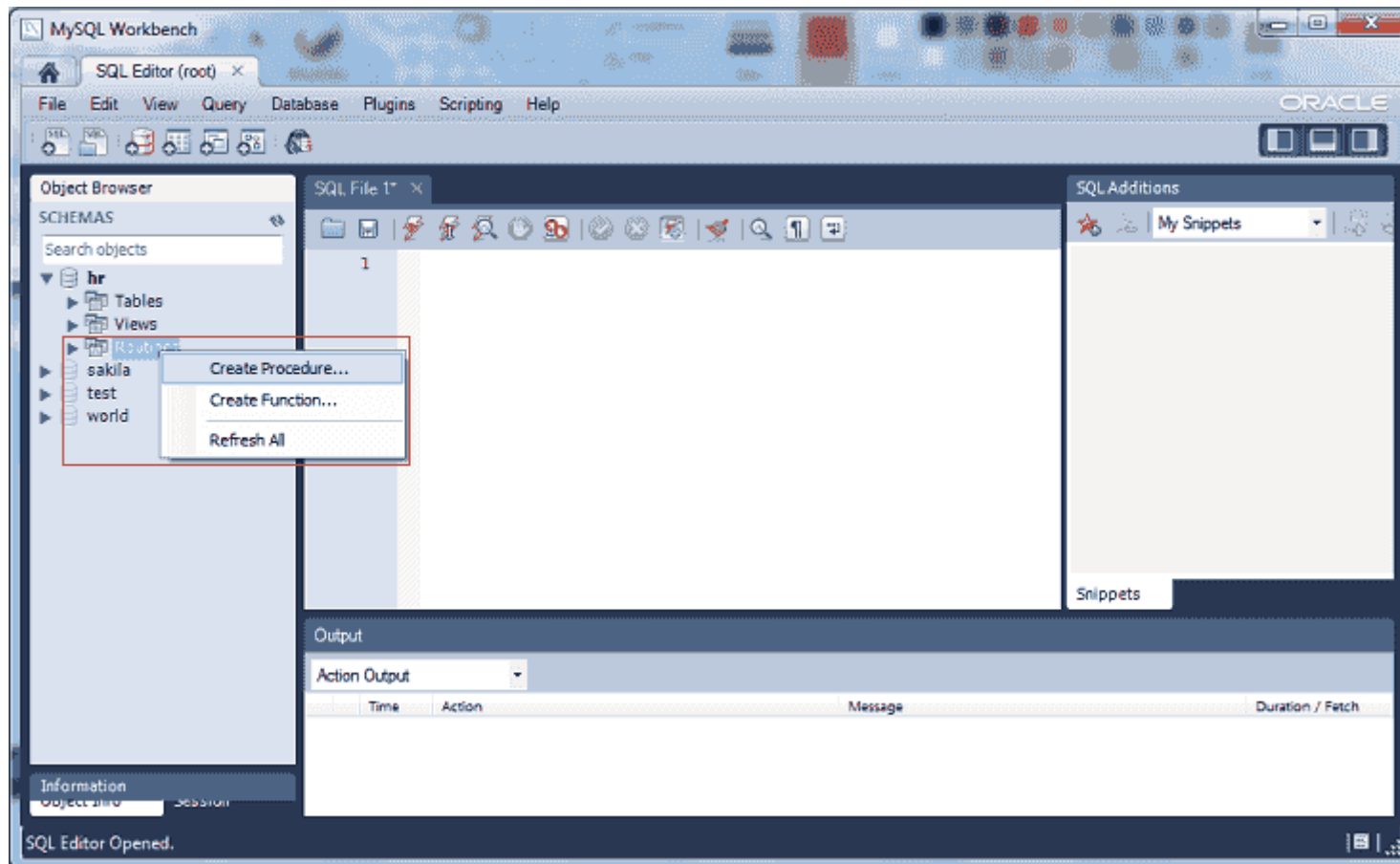
Now input the login details :



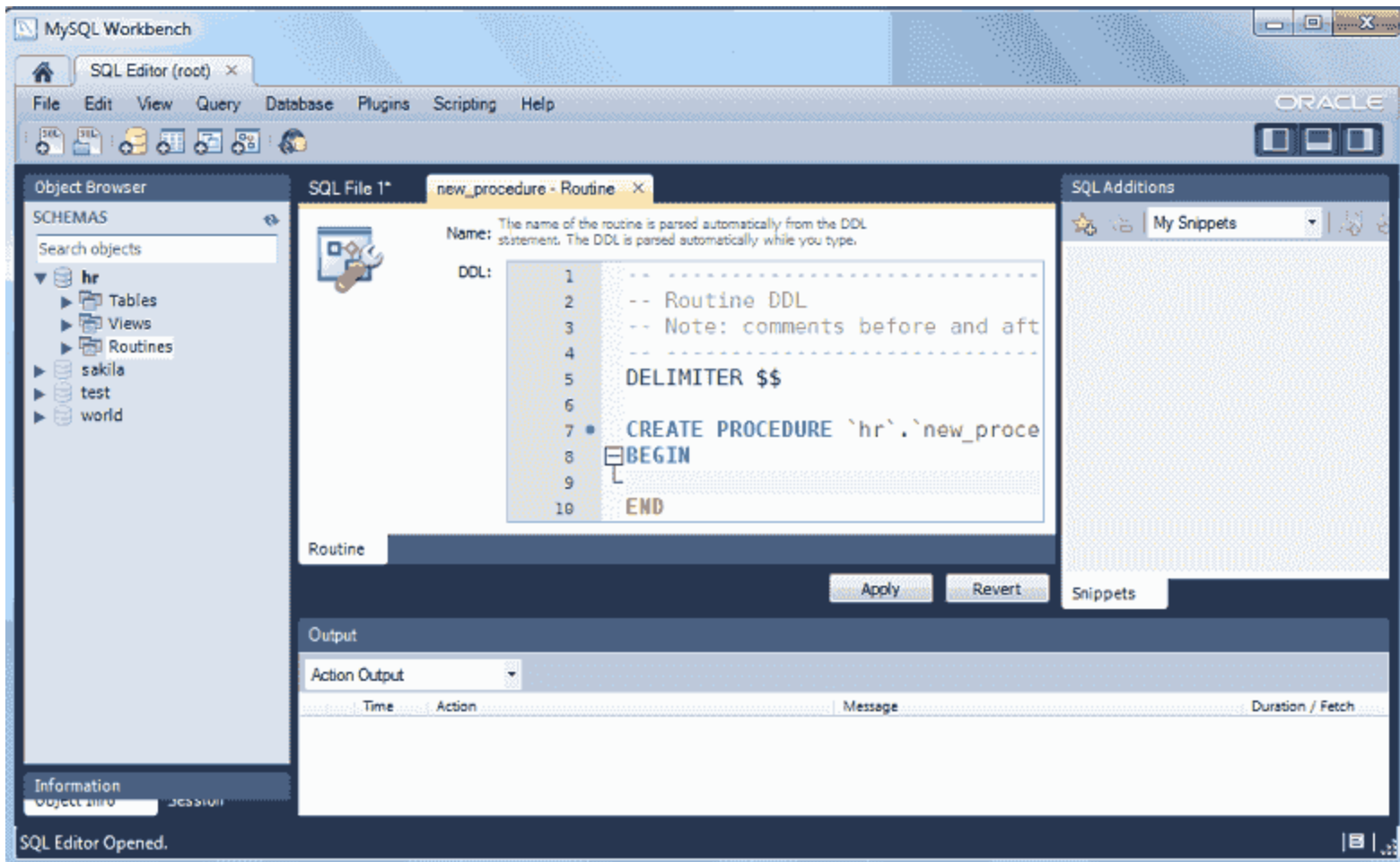
After successful login a new screen will come and from the *object browser panel* select a database :



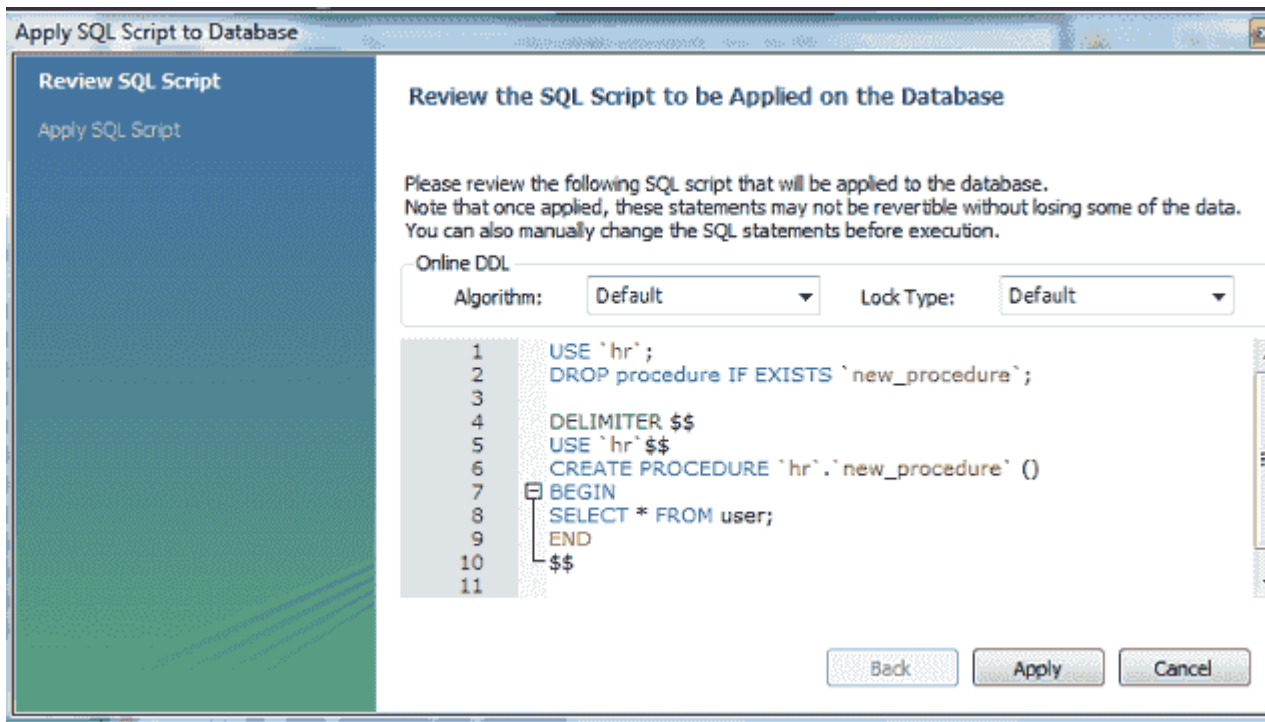
After selecting the database right click on *Routines* a new popup will come :



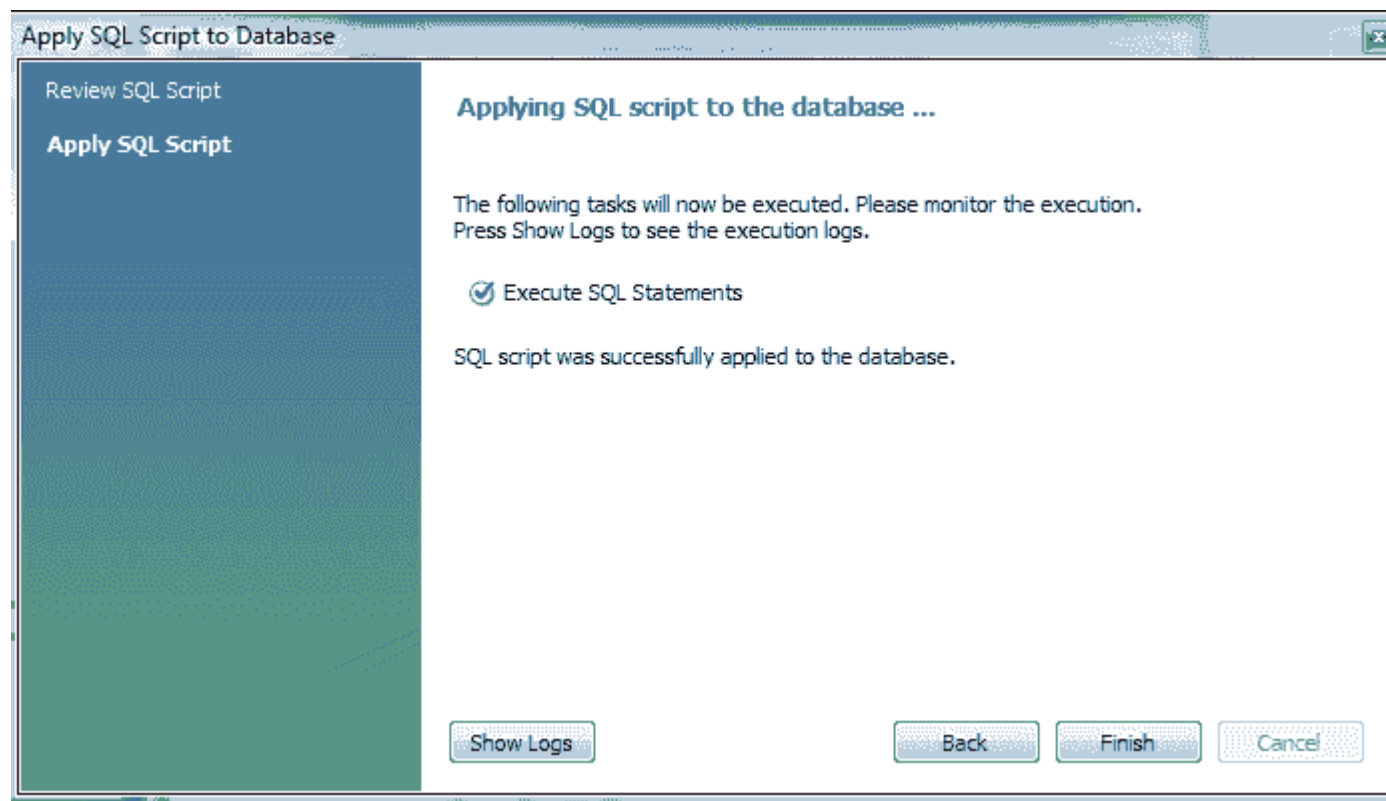
After selecting "Create Procedure" following screen will come where you can write your own procedure.



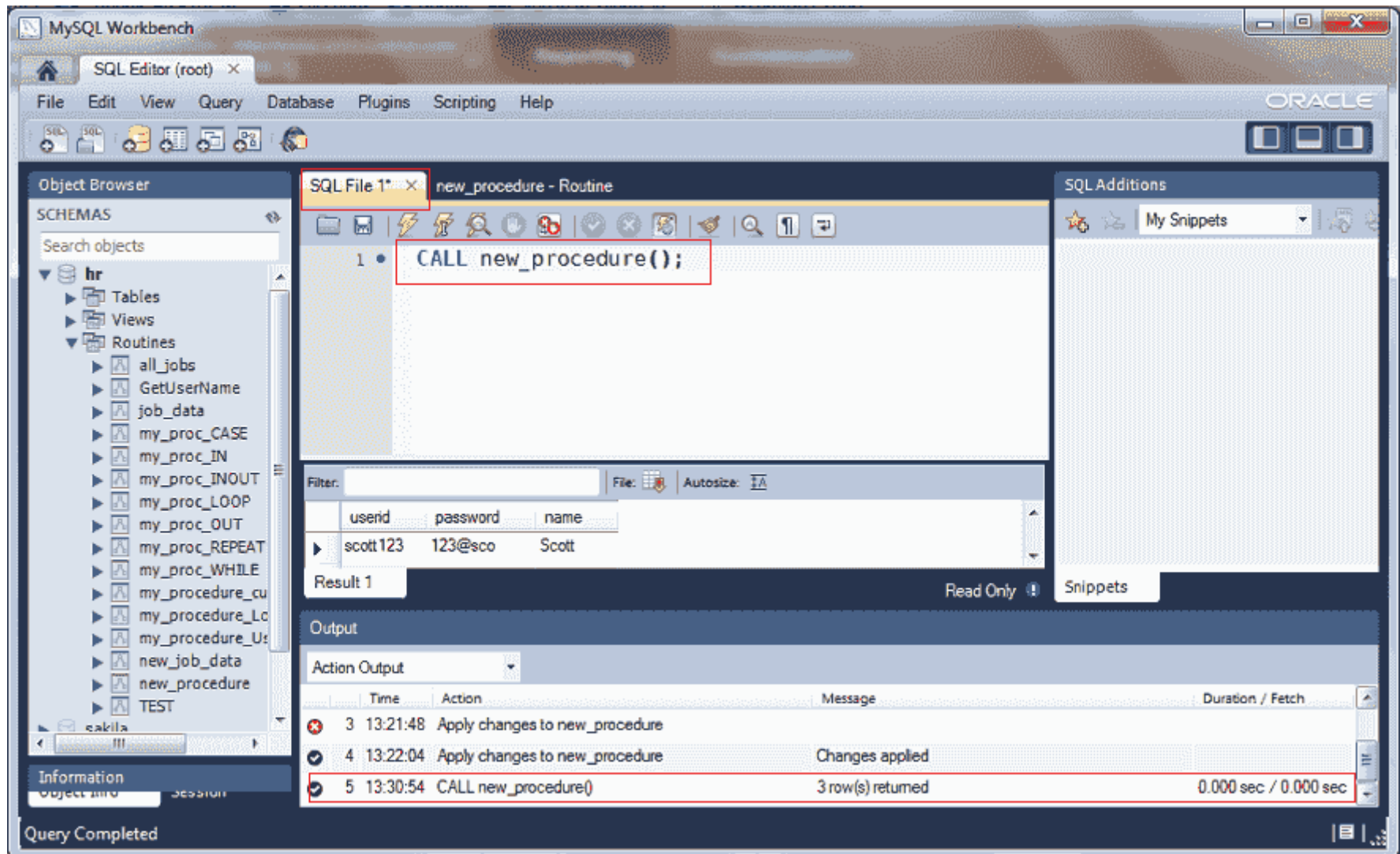
After writing the procedure click on Apply button and the following screen will come :



Next screen will be to review the script and apply on the database.



Now click on Finish button and run the procedure :



Call a procedure

The CALL statement is used to invoke a procedure that is stored in a DATABASE. Here is the syntax :

```
CALL sp_name([parameter[,...]])
```

```
CALL sp_name[()]
```

Stored procedures which do not accept arguments can be invoked without parentheses. Therefore CALL job_data() and CALL job_data are equivalent. Let execute the procedure.

```
mysql> CALL job_data$$
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
AD_PRES	President	20000	40000
AD_VP	Administration Vice President	15000	30000
AD_ASST	Administration Assistant	3000	6000

FI_MGR	Finance Manager	8200	
16000			
FI_ACCOUNT	Accountant	4200	
9000			
AC_MGR	Accounting Manager	8200	
16000			
AC_ACCOUNT	Public Accountant	4200	
9000			
SA_MAN	Sales Manager	10000	
20000			
SA_REP	Sales Representative	6000	
12000			
PU_MAN	Purchasing Manager	8000	
15000			
PU_CLERK	Purchasing Clerk	2500	
5500			
ST_MAN	Stock Manager	5500	
8500			
ST_CLERK	Stock Clerk	2000	
5000			

Let execute the above and see the output :

```
mysql> SHOW CREATE PROCEDURE job_data$$
```

MySQL : Characteristics Clauses

There are some clauses in CREATE PROCEDURE syntax which describe the characteristics of the procedure. The clauses come after the parentheses, but before the body. These clauses are all optional. Here are the clauses :

```
characteristic:
COMMENT 'string'
| LANGUAGE SQL
| [NOT] DETERMINISTIC
| { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
| SQL SECURITY { DEFINER | INVOKER }
```

COMMENT :

The COMMENT characteristic is a MySQL extension. It is used to describe the stored routine and the information is displayed by the SHOW CREATE PROCEDURE statements.

LANGUAGE :

The LANGUAGE characteristic indicates that the body of the procedure is written in SQL.

NOT DETERMINISTIC :

NOT DETERMINISTIC, is informational, a routine is considered "deterministic" if it always produces the same result for the same input parameters, and "not deterministic" otherwise.

CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA

CONTAINS SQL :

CONTAINS SQL means there are no statements that read or write data, in the routine. For example statements SET @x = 1 or DO RELEASE_LOCK('abc'), which execute but neither read nor write data. This is the default if none of these characteristics is given explicitly.

NO SQL :

NO SQL means routine contains no SQL statements.

READS SQL DATA :

READS SQL DATA means the routine contains statements that read data (for example, SELECT), but not statements that write data.

MODIFIES SQL DATA :

MODIFIES SQL DATA means routine contains statements that may write data (for example, INSERT or DELETE).

SQL SECURITY { DEFINER | INVOKER }

SQL SECURITY, can be defined as either SQL SECURITY DEFINER or SQL SECURITY INVOKER to specify the security context; that is, whether the routine executes using the privileges of the account named in the routine DEFINER clause or the user who invokes it. This account must have permission to access the database with which the routine is associated. The default value is DEFINER. The user who invokes the routine must have the EXECUTE privilege for it, as must the DEFINER account if the routine executes in definer security context.

All the above characteristics clauses have defaults. Following two statements produce same result :

```
mysql> CREATE PROCEDURE job_data()  
-> SELECT * FROM JOBS; $$  
Query OK, 0 rows affected (0.00 sec)
```

is the same as :

```
mysql> CREATE PROCEDURE new_job_data()  
-> COMMENT ''  
-> LANGUAGE SQL  
-> NOT DETERMINISTIC  
-> CONTAINS SQL  
-> SQL SECURITY DEFINER  
-> SELECT * FROM JOBS;  
-> $$  
Query OK, 0 rows affected (0.26 sec)
```

In the next section we will discuss on parameters

Before going to MySQL parameters let discuss some MySQL compound statements :

MySQL : Compound-Statement

A compound statement is a block that can contain other blocks; declarations for variables, condition handlers, and cursors; and flow control constructs such as loops and conditional tests. As of version 5.6 MySQL have following compound statements :

- BEGIN ... END Compound-Statement
- Statement Label
- DECLARE
- Variables in Stored Programs
- Flow Control Statements
- Cursors
- Condition Handling

In this section we will discuss the first four statements to cover the parameters part of CREATE PROCEDURE statement.

BEGIN ... END Compound-Statement Syntax

BEGIN ... END block is used to write compound statements, i.e. when you need more than one statement within stored programs (e.g. stored procedures, functions, triggers, and events). Here is the syntax :

```
[begin_label:]
```

```
BEGIN
```

```
[statement_list]
```

END

[end_label])

statement_list : It represents one or more statements terminated by a semicolon(;). The statement_list itself is optional, so the empty compound statement BEGIN END is valid.

begin_label, end_label : See the following section.

Label Statement

Labels are permitted for BEGIN ... END blocks and for the LOOP, REPEAT, and WHILE statements. Here is the syntax :

[begin_label:]

BEGIN

[statement_list]

END [end_label]

[begin_label:]

LOOP

statement_list

END LOOP

[end_label]

[begin_label:]

REPEAT

```
statement_list  
UNTIL search_condition  
END  
REPEAT [end_label]  
[begin_label:]  
WHILE search_condition  
DO  
statement_list  
END WHILE  
[end_label]
```

Label use for those statements which follows following rules:

- *begin_label* must be followed by a colon
- *begin_label* can be given without *end_label*. If *end_label* is present, it must be the same as *begin_label*
- *end_label* cannot be given without *begin_label*.
- Labels at the same nesting level must be distinct
- Labels can be up to 16 characters long.

Declare Statement

The DECLARE statement is used to define various items local to a program, for example local variables, conditions and handlers, cursors. DECLARE is used only inside a BEGIN ...

END compound statement and must be at its start, before any other statements.
Declarations follow the following order :

- Cursor declarations must appear before handler declarations.
- Variable and condition declarations must appear before cursor or handler declarations.

Variables in Stored Programs

System variables and user-defined variables can be used in stored programs, just as they can be used outside stored-program context. Stored programs use DECLARE to define local variables, and stored routines (procedures and functions) can be declared to take parameters that communicate values between the routine and its caller.

Declare a Variable :

```
DECLARE var_name [, var_name] ... type [DEFAULT value]
```

To provide a default value for a variable, include a DEFAULT clause. The value can be specified as an expression; it need not be constant. If the DEFAULT clause is missing, the initial value is NULL.

Example : Local variables

Local variables are declared within stored procedures and are only valid within the BEGIN...END block where they are declared. Local variables can have any SQL data type. The following example shows the use of local variables in a stored procedure.

```
DELIMITER $$  
  
CREATE PROCEDURE my_procedure_Local_Variables()  
BEGIN    /* declare local variables */
```

```

DECLARE a INT DEFAULT 10;
DECLARE b, c INT;      /* using the local variables */
SET a = a + 100;
SET b = 2;
SET c = a + b;
BEGIN      /* local variable in nested block */
DECLARE c INT;
SET c = 5;
/* local variable c takes precedence over the one of the
same name declared in the enclosing block. */
SELECT a, b, c;
END;
SELECT a, b, c;
END$$

```

Now execute the procedure :

```
mysql> CALL my_procedure_Local_Variables();
```

```

+-----+-----+-----+
| a      | b      | c      |

```

```
+-----+-----+-----+
|  110  |      2  |      5  |
+-----+-----+-----+
1 row in set (0.00 sec)
```

```
+-----+-----+-----+
|  a     |  b     |  c     |
+-----+-----+-----+
|  110  |      2  |  112  |
+-----+-----+-----+
1 row in set (0.01 sec)
```

Query OK, 0 rows affected (0.03 sec)

Example : User variables

In MySQL stored procedures, user variables are referenced with an ampersand (@) prefixed to the user variable name (for example, @x and @y). The following example shows the use of user variables within the stored procedure :

```
DELIMITER $$

CREATE PROCEDURE my_procedure_User_Variables()
```

```
BEGIN
SET @x = 15;
SET @y = 10;
SELECT @x, @y, @x-@y;
END$$
```

Now execute the procedure :

```
mysql> CALL my_procedure_User_Variables() ;
```

```
+-----+-----+-----+
| @x    | @y    | @x-@y |
+-----+-----+-----+
|    15 |    10 |     5 |
+-----+-----+-----+
```

```
1 row in set (0.04 sec)
```

```
Query OK, 0 rows affected (0.05 sec)
```

MySQL : Procedure Parameters

Here is the parameter part of CREATE PROCEDURE syntax :

CREATE

```
[DEFINER = { user | CURRENT_USER }]
```

```
PROCEDURE sp_name ([proc_parameter[,...]])
```

```
[characteristic ...] routine_body
```

```
proc_parameter: [ IN | OUT | INOUT ] param_name type
```

We can divide the above CREATE PROCEDURE statement in the following ways :

1. CREATE PROCEDURE sp_name () ...
2. CREATE PROCEDURE sp_name ([IN] param_name type)...
3. CREATE PROCEDURE sp_name ([OUT] param_name type)...
4. CREATE PROCEDURE sp_name ([INOUT] param_name type)...

In the first example, the parameter list is empty.

In the second example, an IN parameter passes a value into a procedure. The procedure might modify the value, but the modification is not visible to the caller when the procedure returns.

In the third example, an OUT parameter passes a value from the procedure back to the

caller. Its initial value is NULL within the procedure, and its value is visible to the caller when the procedure returns.

In the fourth example, an INOUT parameter is initialized by the caller, can be modified by the procedure, and any change made by the procedure is visible to the caller when the procedure returns.

In a procedure, each parameter is an IN parameter by default. To specify otherwise for a parameter, use the keyword OUT or INOUT before the parameter name.

MySQL Procedure : Parameter IN example

In the following procedure, we have used a IN parameter 'var1' (type integer) which accept a number from the user. Within the body of the procedure, there is a SELECT statement which fetches rows from 'jobs' table and the number of rows will be supplied by the user. Here is the procedure :

```
mysql> CREATE PROCEDURE my_proc_IN (IN var1 INT)
-> BEGIN
-> SELECT * FROM jobs LIMIT var1;
-> END$$
Query OK, 0 rows affected (0.00 sec)
```

To execute the first 2 rows from the 'jobs' table execute the following command :

```
mysql> CALL my_proc_in(2) $$
```

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

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
AD_PRES	President	20000	40000
AD_VP	Administration Vice President	15000	30000

2 rows in set (0.00 sec)Query OK, 0 rows affected (0.03 sec)

Now execute the first 5 rows from the 'jobs' table :

```
mysql>
CALL my_proc_in(5)$$
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
--------	-----------	------------	------------

AD_PRES	President	20000	40
AD_VP	Administration Vice President	15000	30
AD_ASST	Administration Assistant	3000	6
FI_MGR	Finance Manager	8200	16
FI_ACCOUNT	Accountant	4200	9

5 rows in set (0.00 sec)Query OK, 0 rows affected (0.05 sec)

MySQL Procedure : Parameter OUT example

The following example shows a simple stored procedure that uses an OUT parameter. Within the procedure MySQL MAX() function retrieves maximum salary from MAX_SALARY of jobs table.

```
mysql> CREATE PROCEDURE my_proc_OUT (OUT highest_salary INT)
-> BEGIN
-> SELECT MAX(MAX_SALARY) INTO highest_salary FROM JOBS;
```

```
-> END$$
```

```
Query OK, 0 rows affected (0.00 sec)
```

In the body of the procedure, the parameter will get the highest salary from MAX_SALARY column. After calling the procedure the word OUT tells the DBMS that the value goes out from the procedure. Here highest_salary is the name of the output parameter and we have passed its value to a session variable named @M, in the CALL statement.

```
mysql> CALL my_proc_OUT(@M) $$
```

```
Query OK, 1 row affected (0.03 sec)
```

```
mysql< SELECT @M$$+-----+
```

```
| @M      |
```

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

```
| 40000   |
```

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

```
1 row in set (0.00 sec)
```

MySQL Procedure : Parameter INOUT example

The following example shows a simple stored procedure that uses an INOUT parameter and an IN parameter. The user will supply 'M' or 'F' through IN parameter (emp_gender) to count a number of male or female from user_details table. The INOUT parameter (mfgender) will return the result to a user. Here is the code and output of the procedure :

```
mysql> CALL my_proc_OUT(@M)$$Query OK, 1 row affected (0.03 sec)mysql> CREATE PROCEDURE my_proc_INOUT (INOUT mfgender INT, IN emp_gender CHAR(1))
-> BEGIN
-> SELECT COUNT(gender) INTO mfgender FROM user_details WHERE gender = emp_gender;
-> END$$
Query OK, 0 rows affected (0.00 sec)
```

Now check the number of **male** and **female** users of the said tables :

```
mysql> CALL my_proc_INOUT(@C, 'M')$$
Query OK, 1 row affected (0.02 sec)
```

```
mysql> SELECT @C$$
+-----+
| @C    |
+-----+
|      3 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> CALL my_proc_INOUT(@C, 'F')$$
Query OK, 1 row affected (0.00 sec)
```

```
mysql> SELECT @C$$
+-----+
| @C    |
+-----+
|      1 |
+-----+
1 row in set (0.00 sec)
```

MySQL : Flow Control Statements

MySQL supports IF, CASE, ITERATE, LEAVE, LOOP, WHILE, and REPEAT constructs for flow control within stored programs. It also supports RETURN within stored functions.

MySQL : If Statement

The IF statement implements a basic conditional construct within a stored programs and must be terminated with a semicolon. There is also an [IF\(\)](#) function, which is different from the IF statement. Here is the syntax of if statement :

```
IF condition THEN statement(s)
[ELSEIF condition THEN statement(s)] ...
[ELSE statement(s)]
END IF
```

- If *the condition* evaluates to true, the corresponding THEN or ELSEIF clause *statements(s)* executes.
- If no *condition* matches, the ELSE clause *statement(s)* executes.
- Each *statement(s)* consists of one or more SQL statements; an empty *statement(s)* is not permitted.

Example :

In the following example, we pass user_id through IN parameter to get the user name. Within the procedure, we have used IF ELSEIF and ELSE statement to get user name against multiple user id. The user name will be stored into INOUT parameter user_name.

```
CREATE DEFINER=`root`@`127.0.0.1`  
PROCEDURE `GetUserName`(INOUT user_name varchar(16),  
IN user_id varchar(16))  
BEGIN  
    DECLARE uname varchar(16);  
    SELECT name INTO uname  
    FROM user  
    WHERE userid = user_id;  
    IF user_id = "scott123"  
    THEN  
        SET user_name = "Scott";  
    ELSEIF user_id = "ferp6734"  
    THEN  
        SET user_name = "Palash";  
    ELSEIF user_id = "diana094"
```

```
THEN
SET user_name = "Diana";
END IF;
END
```

Execute the procedure :

```
mysql> CALL GetUser_name(@A, 'scott123') $$
Query OK, 1 row affected (0.00 sec)
```

```
mysql> SELECT @A;
-> $$
+-----+
| @A    |
+-----+
| Scott |
+-----+
1 row in set (0.00 sec)
```

MySQL : Case Statement

The CASE statement is used to create complex conditional construct within stored programs. The CASE statement cannot have an ELSE NULL clause, and it is terminated with END CASE instead of END. Here is the syntax :

```
CASE case_value  
  
WHEN when_value THEN statement_list  
  
[WHEN when_value THEN statement_list] ...  
  
[ELSE statement_list] END CASE
```

or

```
CASE  
  
WHEN search_condition THEN statement_list  
  
[WHEN search_condition THEN statement_list] ...  
  
[ELSE statement_list] END CASE
```

Explanation : First syntax

- case_value is an expression.
- This value is compared to the when_value expression in each WHEN clause until one of

them is equal.

- When an equal when_value is found, the corresponding THEN clause statement_list executes.

- If no when_value is equal, the ELSE clause statement_list executes, if there is one.

Explanation : Second syntax

- Each WHEN clause search_condition expression is evaluated until one is true, at which point its corresponding THEN clause statement_list executes.

- If no search_condition is equal, the ELSE clause statement_list executes, if there is one.

- Each statement_list consists of one or more SQL statements; an empty statement_list is not permitted.

Example :

We have table called 'jobs' with following records :

+-----+-----+-----+-----+			
-----+			
JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
+-----+-----+-----+-----+			
-----+			
AD_PRES	President	20000	
40000			
AD_VP	Administration Vice President	15000	
30000			

AD_ASST 6000	Administration Assistant	3000
FI_MGR 16000	Finance Manager	8200
FI_ACCOUNT 9000	Accountant	4200
AC_MGR 16000	Accounting Manager	8200
AC_ACCOUNT 9000	Public Accountant	4200
SA_MAN 20000	Sales Manager	10000
SA_REP 12000	Sales Representative	6000
PU_MAN 15000	Purchasing Manager	8000
PU_CLERK 5500	Purchasing Clerk	2500
ST_MAN 8500	Stock Manager	5500

ST_CLERK	Stock Clerk		2000	
5000				
SH_CLERK	Shipping Clerk		2500	
5500				
IT_PROG	Programmer		4000	
10000				
MK_MAN	Marketing Manager		9000	
15000				
MK_REP	Marketing Representative		4000	
9000				
HR_REP	Human Resources Representative		4000	
9000				
PR_REP	Public Relations Representative		4500	
10500				
+-----+-----+-----+-----				
-----+				
19 rows in set (0.03 sec)				

Now we want to count the number of employees with following conditions :

- MIN_SALARY > 10000
- MIN_SALARY < 10000
- MIN_SALARY = 10000

Here is the procedure (the procedure is written into MySQL workbench 5.2 CE) :

```
DELIMITER $$  
CREATE PROCEDURE `hr`.`my_proc_CASE`  
(INOUT no_employees INT, IN salary INT)  
BEGIN  
CASE  
WHEN (salary>10000)  
THEN (SELECT COUNT(job_id) INTO no_employees  
FROM jobs  
WHERE min_salary>10000);  
WHEN (salary<10000)  
THEN (SELECT COUNT(job_id) INTO no_employees  
FROM jobs  
WHERE min_salary<10000);  
ELSE (SELECT COUNT(job_id) INTO no_employees  
FROM jobs WHERE min_salary=10000);  
END CASE;
```

```
END$$
```

In the above procedure, we pass the salary (amount) variable through IN parameter. Within the procedure, there is CASE statement along with two WHEN and an ELSE which will test the condition and return the count value in no_employees. Let execute the procedure in MySQL command prompt :

Number of employees whose salary greater than 10000 :

```
mysql> CALL my_proc_CASE(@C,10001);
```

```
Query OK, 1 row affected (0.00 sec)
```

```
mysql> SELECT @C;
```

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

```
| @C    |
```

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

```
|      2 |
```

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

```
1 row in set (0.00 sec)
```

Number of employees whose salary less than 10000 :

```
mysql> CALL my_proc_CASE(@C,9999);
```

```
Query OK, 1 row affected (0.00 sec)
```

```
mysql> SELECT @C;
```

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

```
| @C      |
```

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

```
|      16 |
```

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

```
1 row in set (0.00 sec)
```

Number of employees whose salary equal to 10000 :

```
mysql> CALL my_proc_CASE(@C,10000);
```

```
Query OK, 1 row affected (0.00 sec)
```

```
mysql> SELECT @C;
```

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

```
| @C      |
```

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

```
|        1 |
```

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

```
1 row in set (0.00 sec)
```

MySQL : ITERATE Statement

ITERATE means "start the loop again". ITERATE can appear only within LOOP, REPEAT, and WHILE statements. Here is the syntax :

```
ITERATE label
```

MySQL : LEAVE Statement

LEAVE statement is used to exit the flow control construct that has the given label. If the label is for the outermost stored program block, LEAVE exits the program. LEAVE can be used within BEGIN ... END or loop constructs (LOOP, REPEAT, WHILE). Here is the syntax :

```
LEAVE label
```

MySQL : LOOP Statement

LOOP is used to create repeated execution of the statement list. Here is the syntax :

```
[begin_label:]
```

```
LOOP
```

```
statement_list
```

```
END LOOP
```

```
[end_label]
```


statement_list consists one or more statements, each statement terminated by a semicolon (;). the statements within the loop are repeated until the loop is terminated. Usually, LEAVE statement is used to exit the loop construct. Within a stored function, RETURN can also be used, which exits the function entirely. A LOOP statement can be labeled.

Example :

In the following procedure rows will be inserted in 'number' table until x is less than num (number supplied by the user through IN parameter). A random number will be stored every time.

```
DELIMITER $$  
CREATE PROCEDURE `my_proc_LOOP` (IN num INT)  
BEGIN  
    DECLARE x INT;  
    SET x = 0;  
loop_label: LOOP  
    INSERT INTO number VALUES (rand());  
    SET x = x + 1;  
    IF x >= num  
    THEN  
        LEAVE loop_label;
```

```
END IF;  
END LOOP;  
END$$
```

Now execute the procedure :

```
mysql> CALL my_proc_LOOP(3);
```

Query OK, 1 row affected, 1 warning (0.19 sec)

```
mysql> select * from number;
```

```
+-----+  
| rnumber      |  
+-----+  
| 0.1228974146 |  
| 0.2705919913 |  
| 0.9842677433 |  
+-----+
```

3 rows in set (0.00 sec)

MySQL : REPEAT Statement

The REPEAT statement executes the statement(s) repeatedly as long as the condition is true. The condition is checked every time at the end of the statements.

```
[begin_label:]  
REPEAT  
statement_list  
UNTIL search_condition  
END  
REPEAT  
[end_label]
```

statement_list : List of one or more statements, each statement terminated by a semicolon(;;).

search_condition : An expression.

A REPEAT statement can be labeled.

Example :

Even numbers are numbers that can be divided evenly by 2. In the following procedure an user passes a number through IN parameter and make a sum of even numbers between 1 and that particular number.

```
DELIMITER $$  
CREATE PROCEDURE my_proc_REPEAT (IN n INT)  
BEIG
```

```

NSET @sum = 0;
SET @x = 1;
REPEAT
IF mod(@x, 2) = 0
THEN
SET @sum = @sum + @x;
END IF;
SET @x = @x + 1;
UNTIL @x > n
END REPEAT;
END $$

```

Now execute the procedure :

```

mysql> call my_proc_REPEAT(5);
Query OK, 0 rows affected (0.00 sec)

```

```

mysql> SELECT @sum;
+-----+
| @sum |

```

```
+-----+
|      6 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> call my_proc_REPEAT(10);
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SELECT @sum;
+-----+
| @sum |
+-----+
|    30 |
+-----+
1 row in set (0.00 sec)
```

MySQL : RETURN Statement

The RETURN statement terminates execution of a stored function and returns the value *expr* to the function caller. There must be at least one RETURN statement in a stored

function. There may be more than one if the function has multiple exit points. Here is the syntax :

```
RETURN expr
```

This statement is not used in stored procedures or triggers. The LEAVE statement can be used to exit a stored program of those types.

MySQL : WHILE Statement

The WHILE statement executes the statement(s) as long as the condition is true. The condition is checked every time at the beginning of the loop. Each statement is terminated by a semicolon (;). Here is the syntax :

```
[begin_label:] WHILE search_condition DO
```

```
    statement_list
```

```
END WHILE [end_label]
```

A WHILE statement can be labeled.

Example :

Odd numbers are numbers that cannot be divided exactly by 2. In the following procedure, a user passes a number through IN parameter and make a sum of odd numbers between 1 and that particular number.

```
DELIMITER $$
```

```
CREATE PROCEDURE my_proc_WHILE(IN n INT)
```

```

BEGIN
SET @sum = 0;
SET @x = 1;
WHILE @x<n
DO
    IF mod(@x, 2) <> 0 THEN
SET @sum = @sum + @x;
END IF;
SET @x = @x + 1;
END WHILE;
END$$

```

Now execute the procedure :

```
mysql> CALL my_proc_WHILE(5);
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SELECT @sum;
```

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

```
| @sum |
```

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

```
|      3 |
```

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

```
1 row in set (0.00 sec)
```

```
mysql> CALL my_proc_WHILE(10);
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SELECT @sum;
```

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

```
| @sum |
```

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

```
| 25 |
```

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

```
1 row in set (0.00 sec)
```

```
mysql> CALL my_proc_WHILE(3);
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SELECT @sum;
```



```
+-----+
| @sum |
+-----+
|      4 |
+-----+
1 row in set (0.00 sec)
```

MySQL : ALTER PROCEDURE

This statement can be used to change the characteristics of a stored procedure. More than one change may be specified in an ALTER PROCEDURE statement. However, you cannot change the parameters or body of a stored procedure using this statement; to make such changes, you must drop and re-create the procedure using DROP PROCEDURE and CREATE PROCEDURE. Here is the syntax :

```
ALTER PROCEDURE proc_name [characteristic ...]characteristic:
COMMENT 'string'
| LANGUAGE SQL
| { CONTAINS SQL
| NO SQL | READS SQL DATA
| MODIFIES SQL DATA }
| SQL SECURITY { DEFINER
```

```
| INVOKER }
```

You must have the ALTER ROUTINE privilege for the procedure. By default, that privilege is granted automatically to the procedure creator. In our previous procedure "my_proc_WHILE" the comment section was empty. To input new comment or modify the previous comment use the following command :

```
mysql> ALTER PROCEDURE my_proc_WHILE  
COMMENT 'Modify Comment';  
>Query OK, 0 rows affected (0.20 sec)
```

You can check the result through SHOW CREATE PROCEDURE command which we have discussed earlier.

MySQL : DROP PROCEDURE

This statement is used to drop a stored procedure or function. That is, the specified routine is removed from the server. You must have the ALTER ROUTINE privilege for the routine. (If the automatic_sp_privileges system variable is enabled, that privilege and EXECUTE are granted automatically to the routine creator when the routine is created and dropped from the creator when the routine is dropped)

```
DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp_name
```

The IF EXISTS clause is a MySQL extension. It prevents an error from occurring if the procedure or function does not exist. A warning is produced that can be viewed with SHOW WARNINGS. Here is an example :

```
mysql> DROP PROCEDURE new_procedure;
```

```
Query OK, 0 rows affected (0.05 sec)
```

You can check the result through SHOW CREATE PROCEDURE command which we have discussed earlier.

MySQL : Cursors (LIKE EXCEL ROWS)

A database cursor is a control structure that enables traversal over the records in a database. Cursors are used by database programmers to process individual rows returned by database system queries. Cursors enable manipulation of whole result sets at once. In this scenario, a cursor enables the rows in a result set to be processed sequentially. In SQL procedures, a cursor makes it possible to define a result set (a set of data rows) and perform complex logic on a row by row basis. By using the same mechanics, an SQL procedure can also define a result set and return it directly to the caller of the SQL procedure or to a client application.

MySQL supports cursors inside stored programs. The syntax is as in embedded SQL. Cursors have these properties :

- Asensitive: The server may or may not make a copy of its result table
- Read only: Not updatable
- Nonscrollable: Can be traversed only in one direction and cannot skip rows

To use cursors in MySQL procedures, you need to do the following :

- Declare a cursor.
- Open a cursor.
- Fetch the data into variables.
- Close the cursor when done.

Declare a cursor :

The following statement declares a cursor and associates it with a SELECT statement that retrieves the rows to be traversed by the cursor.

```
DECLARE cursor_name  
CURSOR FOR select_statement
```

Open a cursor :

The following statement opens a previously declared cursor.

```
OPEN cursor_name
```

Fetch the data into variables :

This statement fetches the next row for the SELECT statement associated with the specified cursor (which must be open) and advances the cursor pointer. If a row exists, the fetched columns are stored in the named variables. The number of columns retrieved by the SELECT statement must match the number of output variables specified in the FETCH statement.

```
FETCH [[NEXT] FROM] cursor_name  
INTO var_name [, var_name] ...
```

Close the cursor when done :

This statement closes a previously opened cursor. An error occurs if the cursor is not open.

```
CLOSE cursor_name
```

Example :

The procedure starts with three variable declarations. Incidentally, the order is important. First, declare variables. Then declare conditions. Then declare cursors. Then, declare handlers. If you put them in the wrong order, you will get an error message.

```
DELIMITER $$  
  
CREATE PROCEDURE my_procedure_cursors(INOUT return_val INT)  
BEGIN  
    DECLARE a,b INT;  
    DECLARE cur_1 CURSOR FOR  
    SELECT max_salary FROM jobs;  
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET b = 1;  
    OPEN cur_1;REPEATFETCH cur_1 INTO a;  
    UNTIL b = 1END REPEAT;  
    CLOSE cur_1;  
    SET return_val = a;  
END;  
$$
```

Now execute the procedure :

```
mysql>
```

```
CALL my_procedure_cursors (@R);  
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SELECT @R;  
+-----+  
| @R      |  
+-----+  
| 10500   |  
+-----+  
1 row in set (0.00 sec)
```

We will provide more examples on cursors soon.

MySQL Triggers

Introduction on Triggers

A trigger is a set of actions that are run automatically when a specified change operation (SQL INSERT, UPDATE, or DELETE statement) is performed on a specified table. Triggers are useful for tasks such as enforcing business rules, validating input data, and keeping an audit trail.

Uses for triggers :

- Enforce business rules
- Validate input data
- Generate a unique value for a newly-inserted row in a different file.
- Write to other files for audit trail purposes
- Query from other files for cross-referencing purposes
- Access system functions
- Replicate data to different files to achieve data consistency

Benefits of using triggers in business :

- Faster application development. Because the database stores triggers, you do not have to code the trigger actions into each database application.
- Global enforcement of business rules. Define a trigger once and then reuse it for any application that uses the database.
- Easier maintenance. If a business policy changes, you need to change only the corresponding trigger program instead of each application program.

- Improve performance in client/server environment. All rules run on the server before the result returns.

Implementation of SQL triggers is based on the SQL standard. It supports constructs that are common to most programming languages. It supports the declaration of local variables, statements to control the flow of the procedure, assignment of expression results to variables, and error handling.

MySQL Triggers

We assume that you are habituated with "MySQL Stored Procedures", if not you can read our [MySQL Procedures](#) tutorial. You can use the following statements of MySQL procedure in triggers :

- Compound statements ([BEGIN / END](#))
- Variable declaration ([DECLARE](#)) and assignment (SET)
- Flow-of-control statements
([IF](#), [CASE](#), [WHILE](#), [LOOP](#), [WHILE](#), [REPEAT](#), [LEAVE](#), [ITERATE](#))
- Condition declarations
- Handler declarations

How to create MySQL triggers ?

A trigger is a named database object that is associated with a table, and it activates when a particular event (e.g. an insert, update or delete) occurs for the table. The statement CREATE TRIGGER creates a new trigger in MySQL. Here is the syntax :

Syntax :

[CREATE](#)

[DEFINER = { user | CURRENT_USER }]

TRIGGER trigger_name

trigger_time trigger_event

ON tbl_name FOR EACH ROW

trigger_body

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }

Explanation :

DEFINER clause : The DEFINER clause specifies the MySQL account to be used when checking access privileges at trigger activation time. If a user value is given, it should be a MySQL account specified as 'user_name'@'host_name' (the same format used in the GRANT statement), CURRENT_USER, or CURRENT_USER().

The default DEFINER value is the user who executes the CREATE TRIGGER statement.

This is the same as specifying DEFINER = CURRENT_USER explicitly.

If you specify the DEFINER clause, these rules determine the valid DEFINER user values :

- If you do not have the SUPER privilege, the only permitted user value is your own account, either specified literally or by using CURRENT_USER. You cannot set the definer to some other account.
- If you have the SUPER privilege, you can specify any syntactically valid account name. If the account does not actually exist, a warning is generated.
- Although it is possible to create a trigger with a nonexistent DEFINER account, it is not a good idea for such triggers to be activated until the account actually does exist. Otherwise, the behavior with respect to privilege checking is undefined.

trigger_name : All triggers must have unique names within a schema. Triggers in different schemas can have the same name.

trigger_time : trigger_time is the trigger action time. It can be BEFORE or AFTER to indicate that the trigger activates before or after each row to be modified.

trigger_event : trigger_event indicates the kind of operation that activates the trigger.

These trigger_event values are permitted:

- The trigger activates whenever a new row is inserted into the table; for example, through INSERT, LOAD DATA, and REPLACE statements.
- The trigger activates whenever a row is modified; for example, through UPDATE statements.
- The trigger activates whenever a row is deleted from the table; for example, through DELETE and REPLACE statements. DROP TABLE and TRUNCATE TABLE statements on the table do not activate this trigger, because they do not use DELETE. Dropping a partition does not activate DELETE triggers, either.

tbl_name : The trigger becomes associated with the table named tbl_name, which must refer to a permanent table. You cannot associate a trigger with a TEMPORARY table or a view.

trigger_body : trigger_body is the statement to execute when the trigger activates. To execute multiple statements, use the BEGIN ... END compound statement construct. This also enables you to use the same statements that are permissible within stored routines. Here is a simple example :

```
mysql> CREATE TRIGGER ins_sum BEFORE INSERT ON account
      -> FOR EACH ROW SET @sum = @sum + NEW.amount;
Query OK, 0 rows affected (0.06 sec)
```

In the above example, there is new keyword '**NEW**' which is a MySQL extension to triggers. There is two MySQL extension to triggers '**OLD**' and '**NEW**'. OLD and NEW are not case sensitive.

- Within the trigger body, the OLD and NEW keywords enable you to access columns in the rows affected by a trigger
- In an INSERT trigger, only NEW.col_name can be used.
- In a UPDATE trigger, you can use OLD.col_name to refer to the columns of a row before it is updated and NEW.col_name to refer to the columns of the row after it is updated.
- In a DELETE trigger, only OLD.col_name can be used; there is no new row.

A column named with OLD is read only. You can refer to it (if you have the SELECT privilege), but not modify it. You can refer to a column named with NEW if you have the SELECT privilege for it. In a BEFORE trigger, you can also change its value with SET NEW.col_name = value if you have the UPDATE privilege for it. This means you can use a trigger to modify the values to be inserted into a new row or used to update a row. (Such a SET statement has no effect in an AFTER trigger because the row change will have already occurred.)

Sample database, table, table structure, table records for various examples










Database Name : hr

Host Name : localhost

Database user : root

Password : ''

Structure of the table : **emp_details**

Table Name: <input type="text" value="emp_details"/>		Schema: hr							
Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
 EMPLOYEE_ID	DECIMAL(6,0)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	0
 FIRST_NAME	VARCHAR(20)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
 LAST_NAME	VARCHAR(25)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
 EMAIL	VARCHAR(25)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
 PHONE_NUMBER	VARCHAR(20)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
 HIRE_DATE	DATE	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
 JOB_ID	VARCHAR(10)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
 SALARY	DECIMAL(8,2)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
 COMMISSION_PCT	DECIMAL(2,2)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL

Records of the table (on some fields): **emp_details**

```
mysql> SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, JOB_ID, SALARY, C  
OMMISSION_PCT FROM emp_details;
```

```

+-----+-----+-----+-----+-----+-----+
| EMPLOYEE_ID | FIRST_NAME | LAST_NAME | JOB_ID | SALARY | COMMI |
| SSION_PCT |
+-----+-----+-----+-----+-----+-----+
|          100 | Steven    | King      | AD_PRES | 24000.00 |      |
| 0.10 |
|          101 | Neena     | Kochhar   | AD_VP   | 17000.00 |      |
| 0.50 |
|          102 | Lex       | De Haan   | AD_VP   | 17000.00 |      |
| 0.50 |
|          103 | Alexander | Hunold    | IT_PROG | 9000.00  |      |
| 0.25 |
|          104 | Bruce     | Ernst     | IT_PROG | 6000.00  |      |
| 0.25 |
|          105 | David     | Austin    | IT_PROG | 4800.00  |      |
| 0.25 |
+-----+-----+-----+-----+-----+-----+
6 rows in set (0.00 sec)

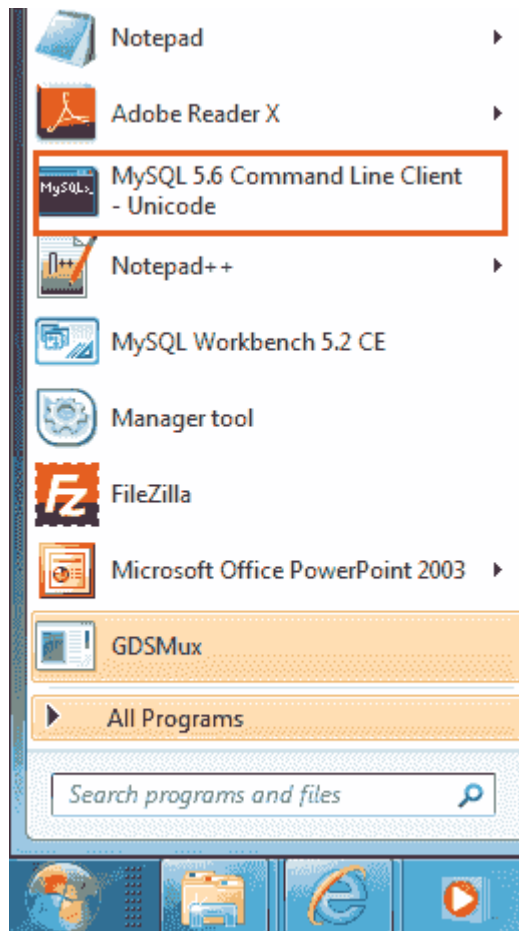
```

Tool to create MySQL Triggers

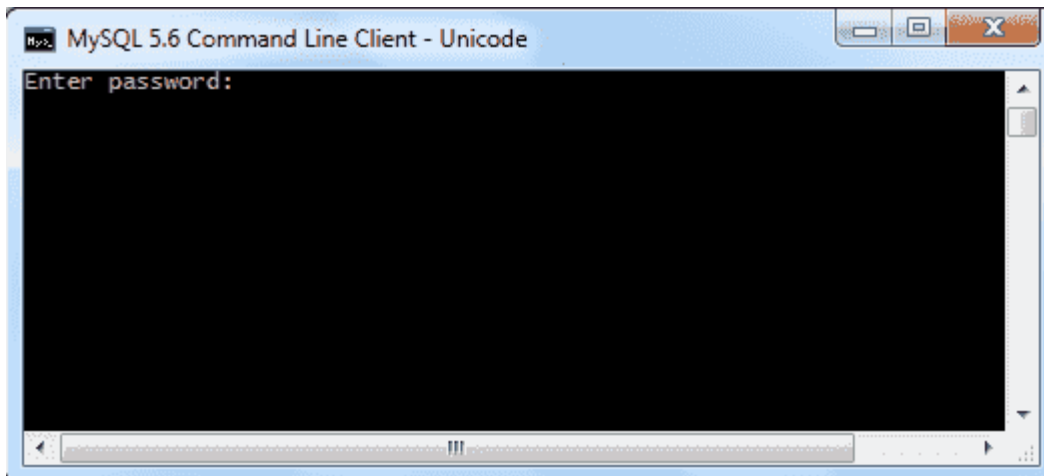
You can write a procedure in MySQL command line tool or you can use [MySQL workbench](#) which is an excellent front-end tool (here we have used version 5.3 CE).

MySQL command line tool : -

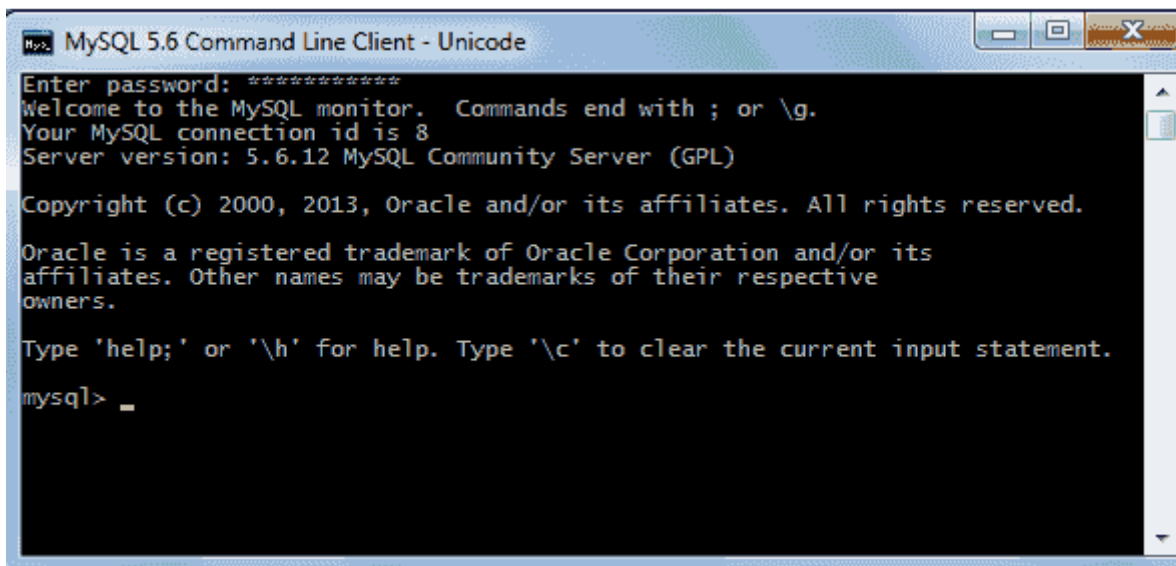
Select MySQL command Client from Start menu :



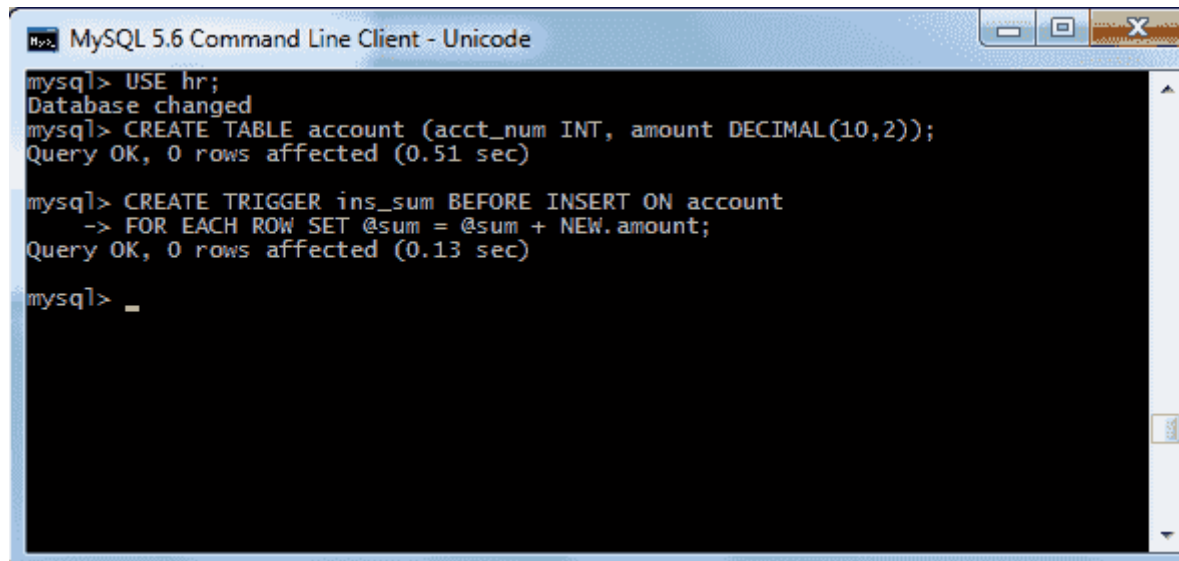
Selecting MySQL command prompt following screen will come :



After a successful login, you can access the MySQL command prompt :



Now you can write your own trigger on a specific table, see the following example :



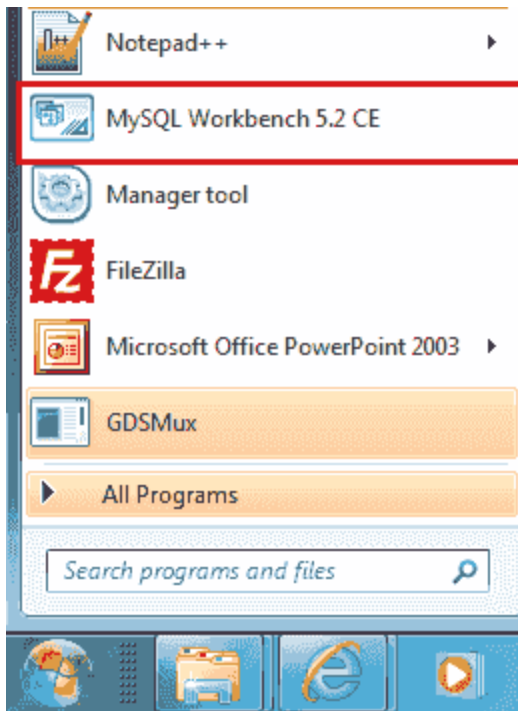
```
mysql> USE hr;
Database changed
mysql> CREATE TABLE account (acct_num INT, amount DECIMAL(10,2));
Query OK, 0 rows affected (0.51 sec)

mysql> CREATE TRIGGER ins_sum BEFORE INSERT ON account
-> FOR EACH ROW SET @sum = @sum + NEW.amount;
Query OK, 0 rows affected (0.13 sec)

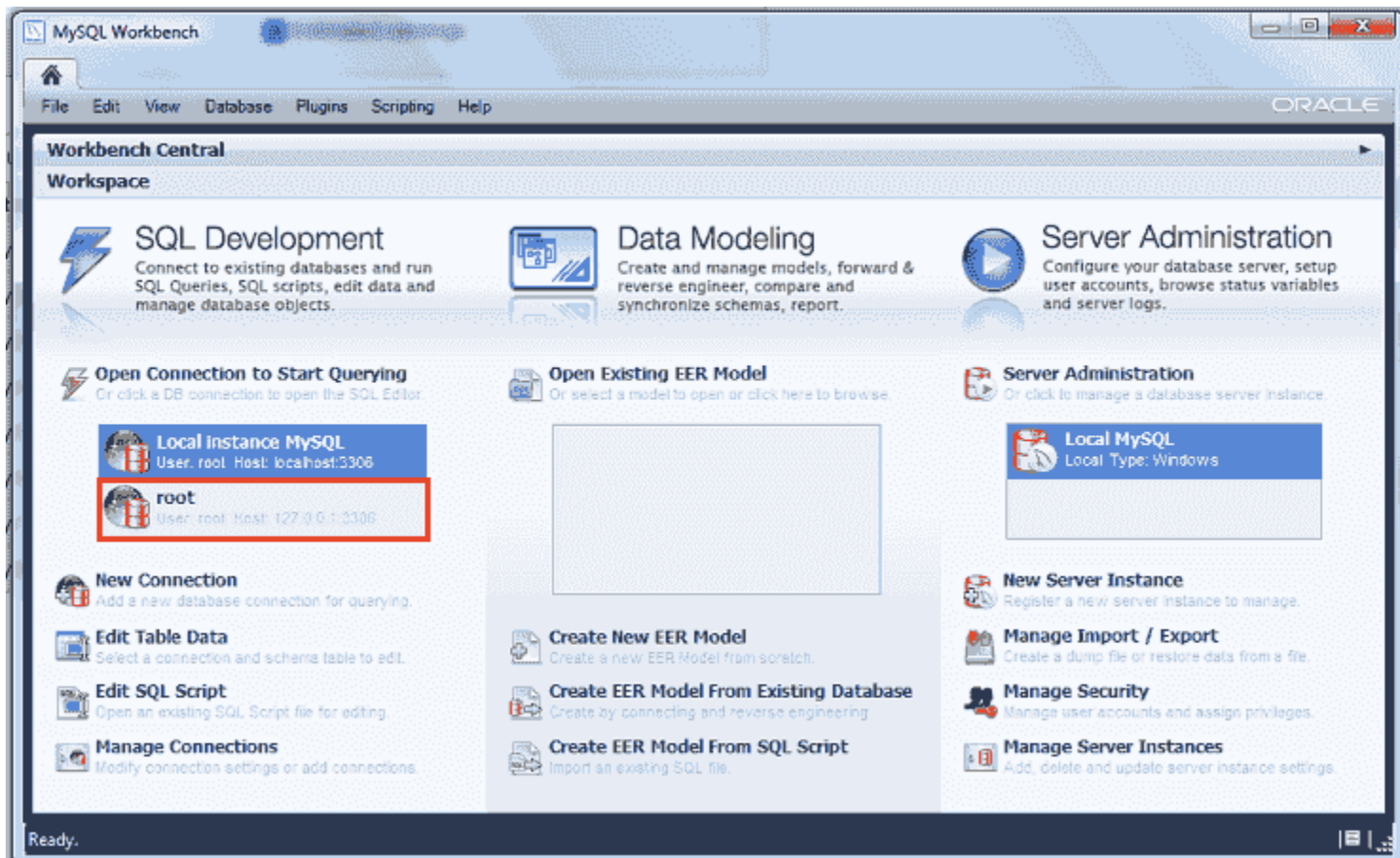
mysql> _
```

MySQL workbench (5.3 CE) : -

Select MySQL workbench from Start menu :



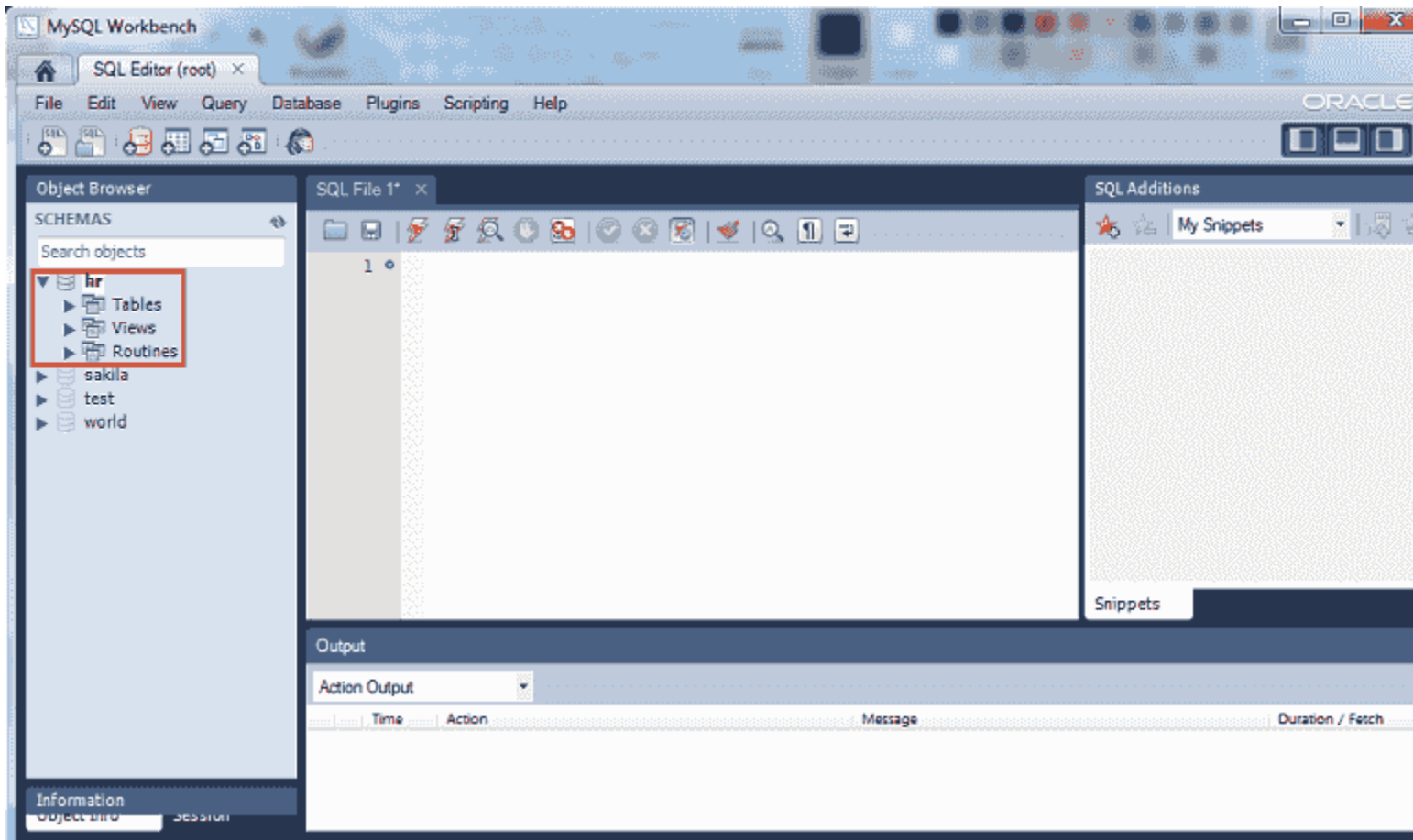
After selecting MySQL workbench following login screen will come :



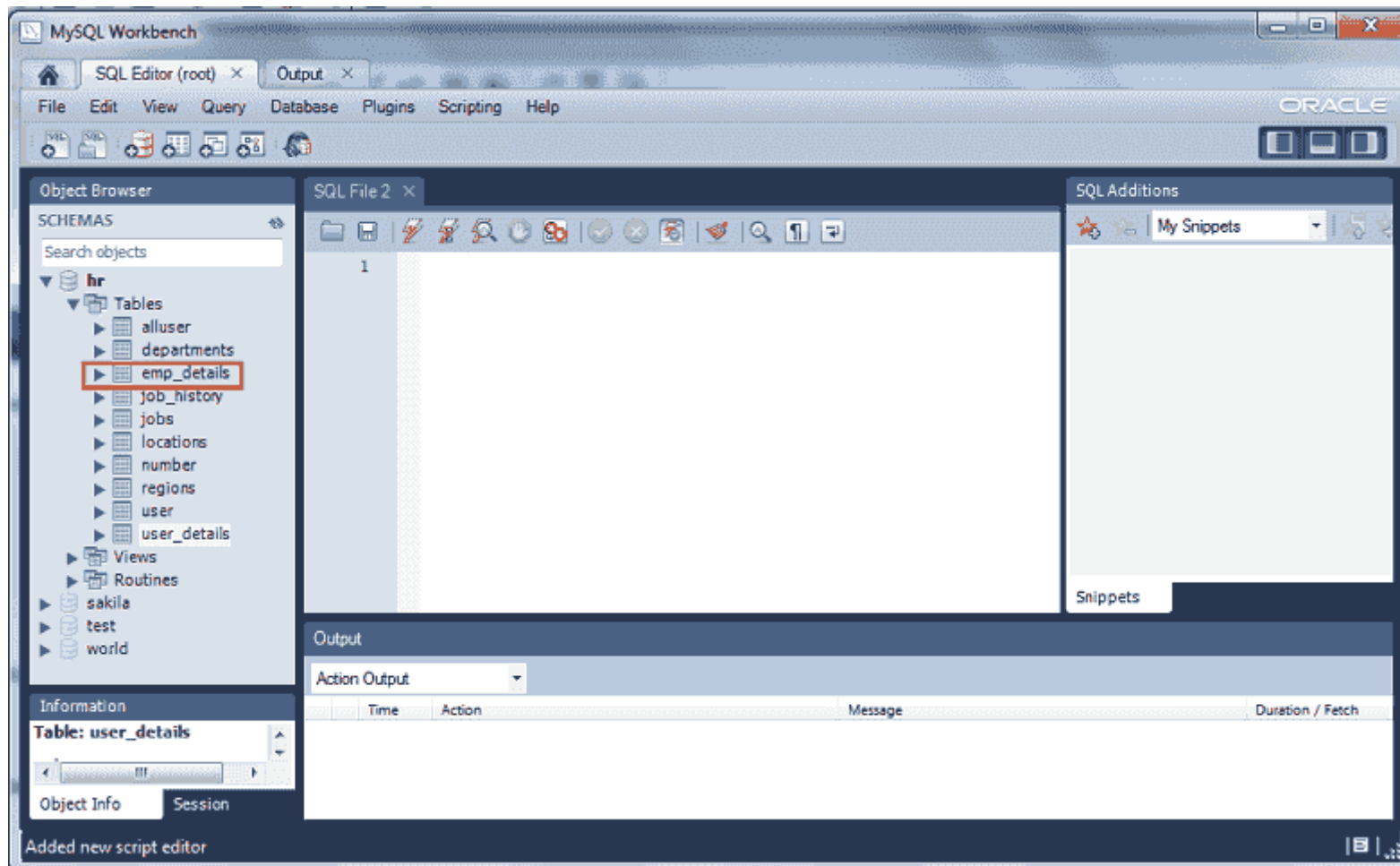
Now input the login details :



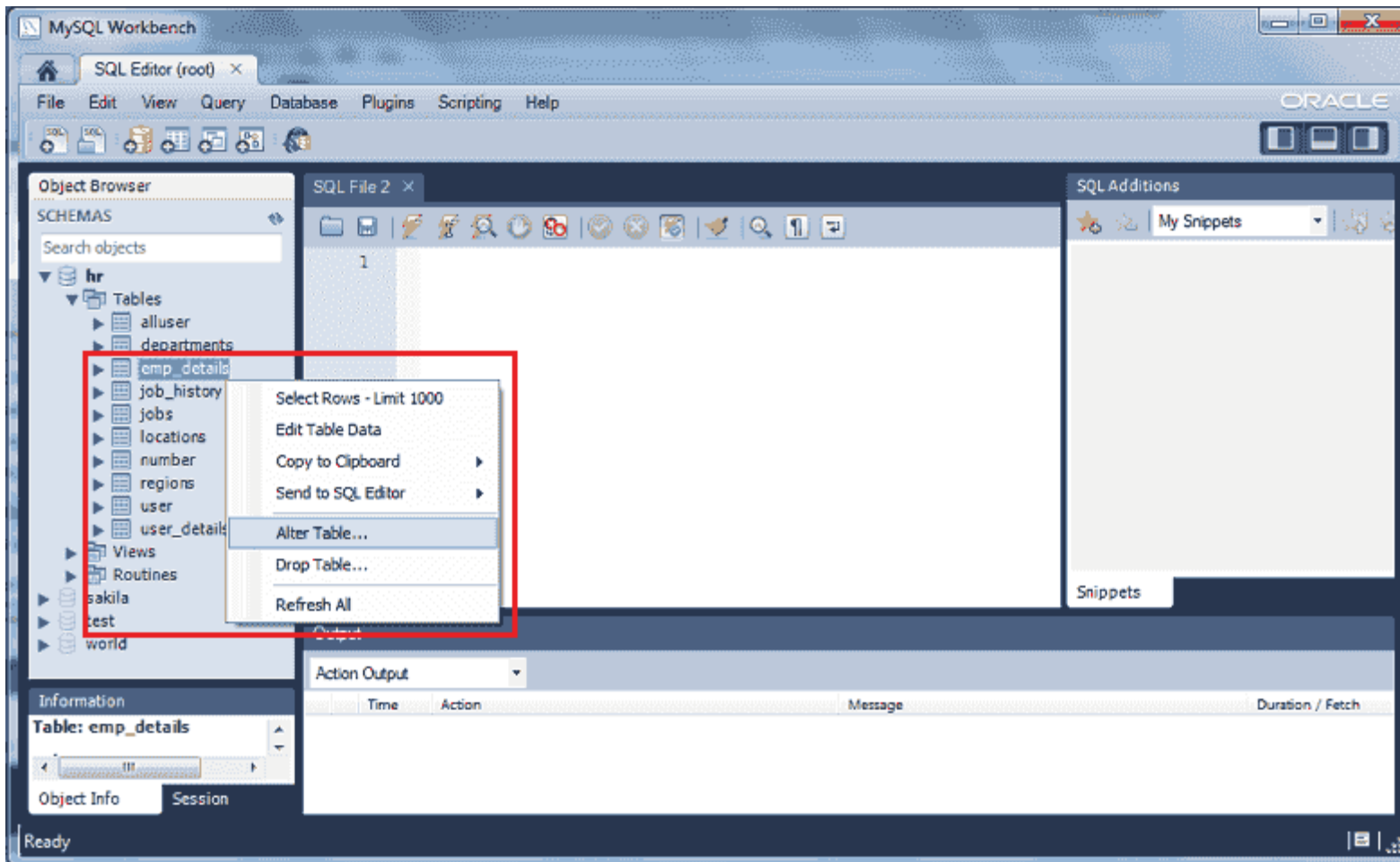
After successful login, a new screen will come and from the *object browser panel* select a database :



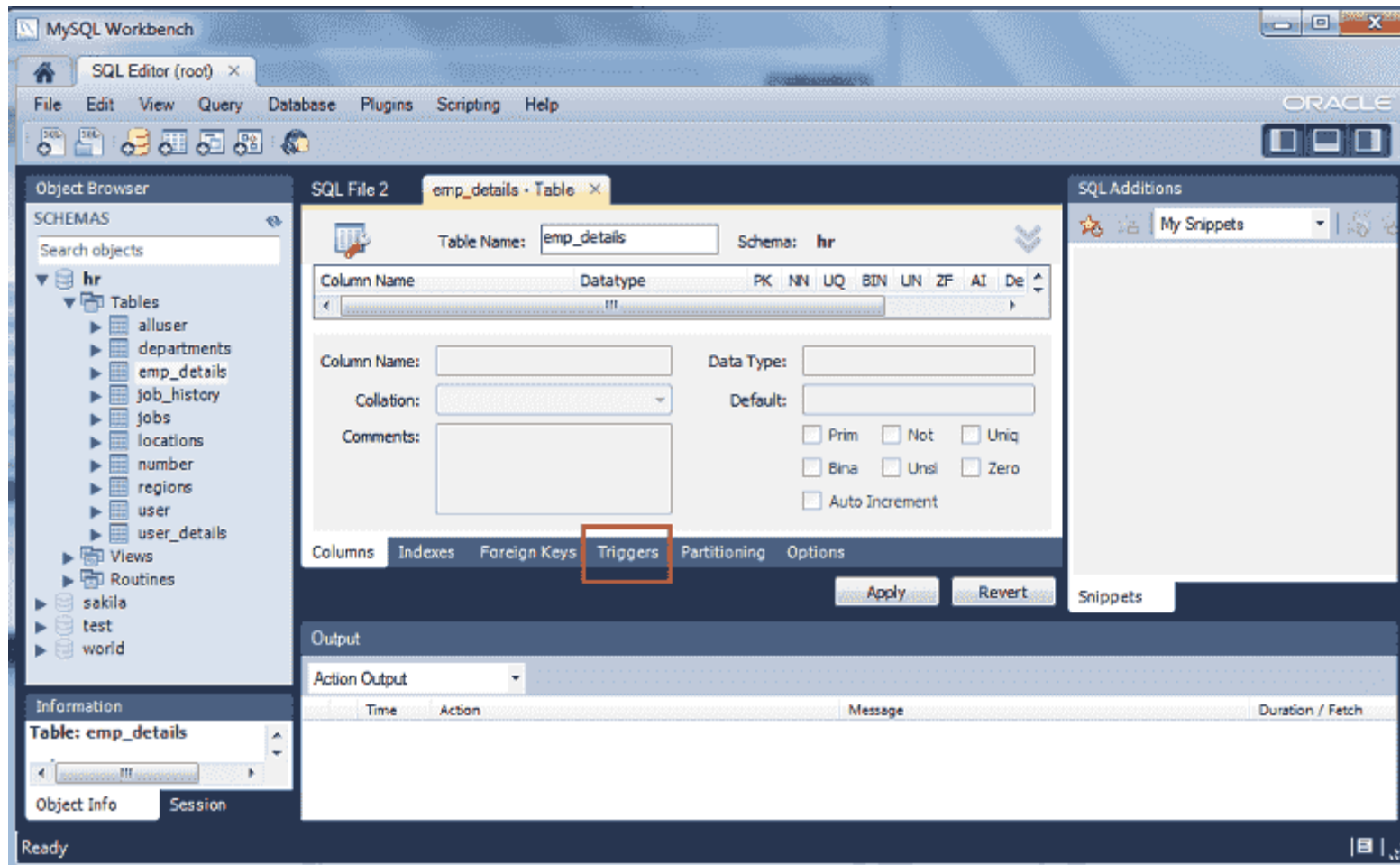
After selecting the database, select the tables :



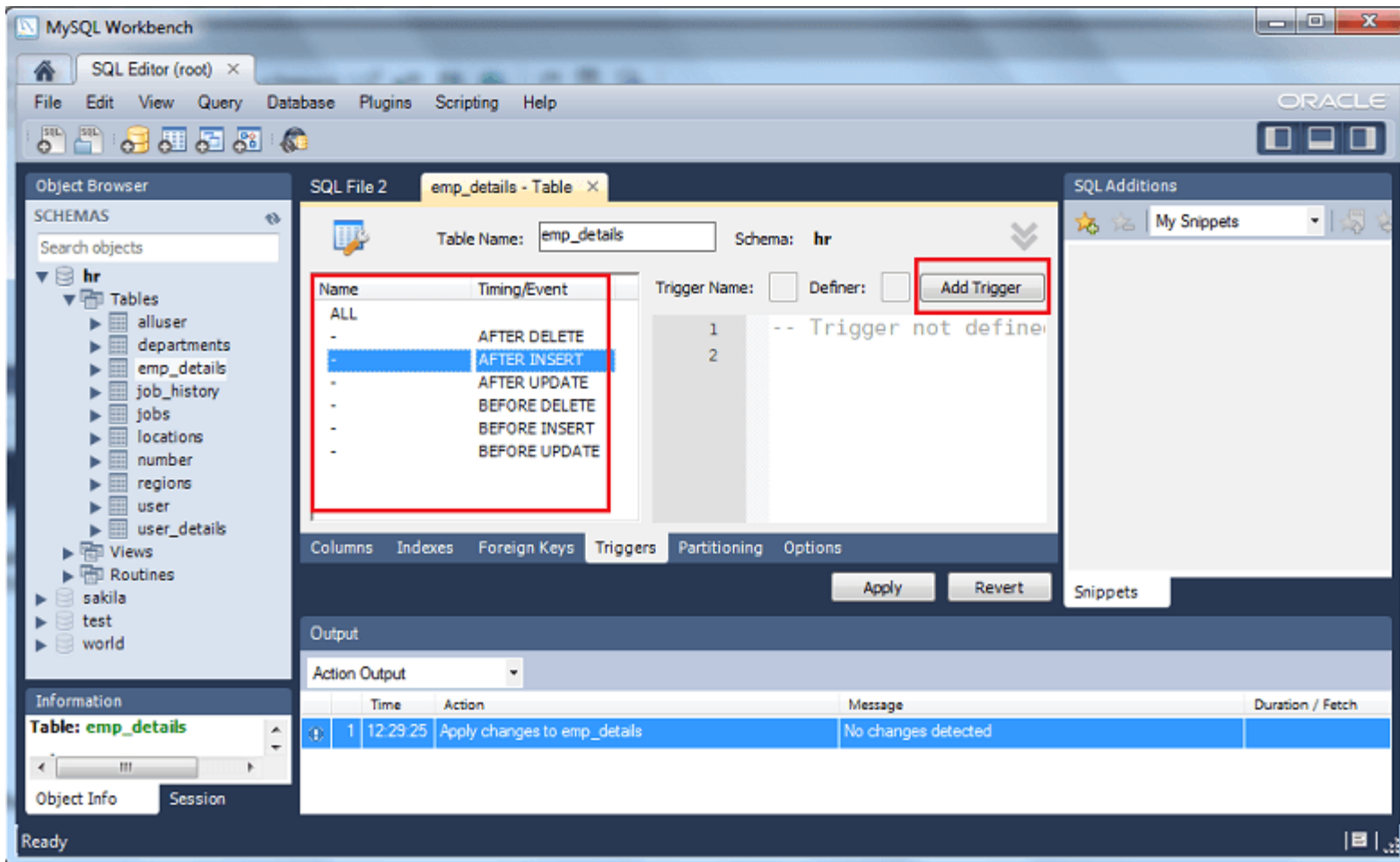
Now right click on emp_details a window pops up, click on *Alter Table* :



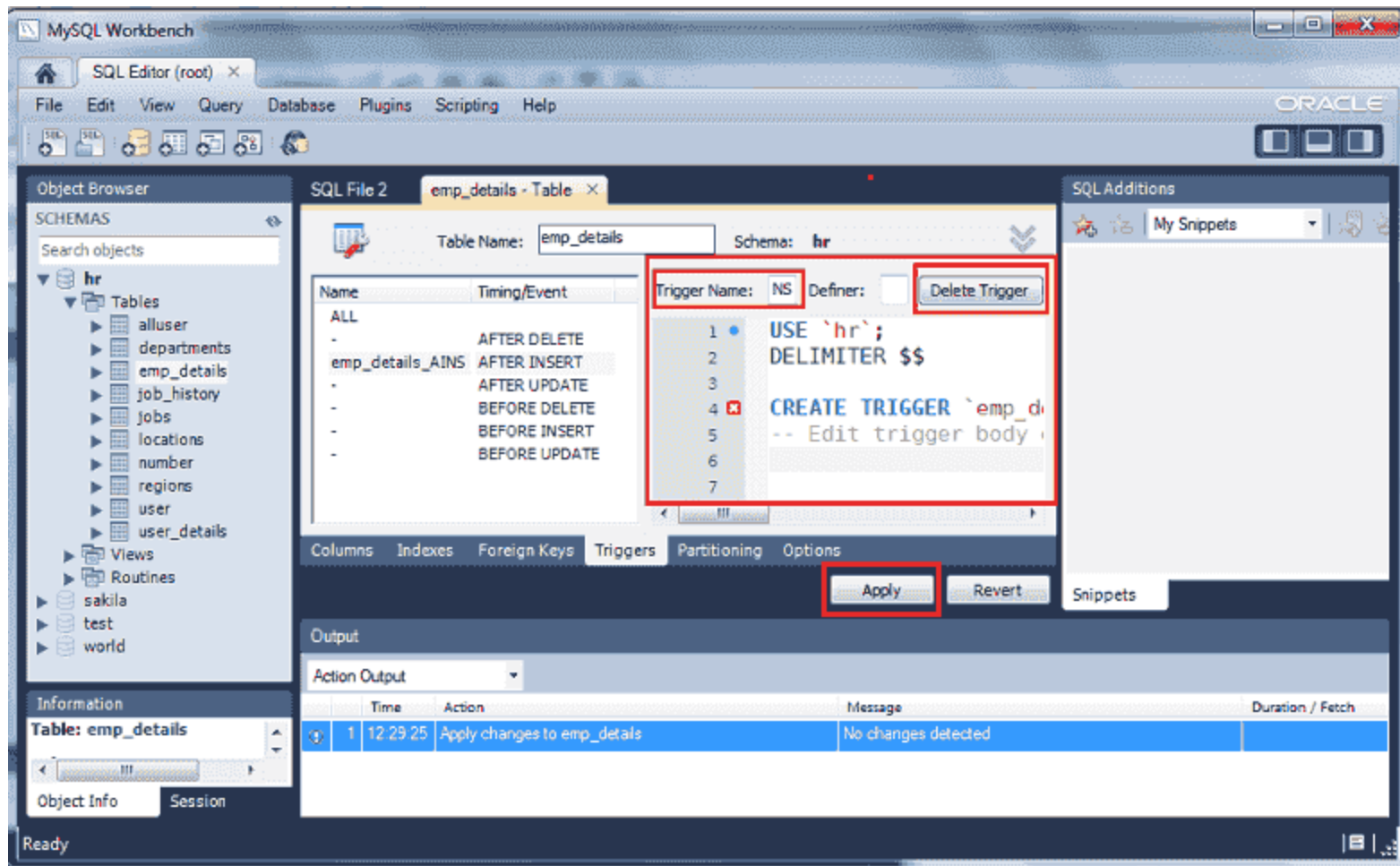
Clicking on " *Alter Table* " details of emp_details will come :



Now click on *Trigger* tab in the previous section, then select the Timing/Event it may be AFTER DELETE, AFTER INSERT, AFTER UPDATE or BEFORE DELETE, BEFORE INSERT OR BEFORE UPDATE. Let we select AFTER INSERT, you also notice that there is a button *Add Trigger*.



Clicking on *Add Trigger* button a default code on trigger will come on the basis of choosing Timing/Event :



Trigger Name : emp_details_AINS

Default Trigger code details :

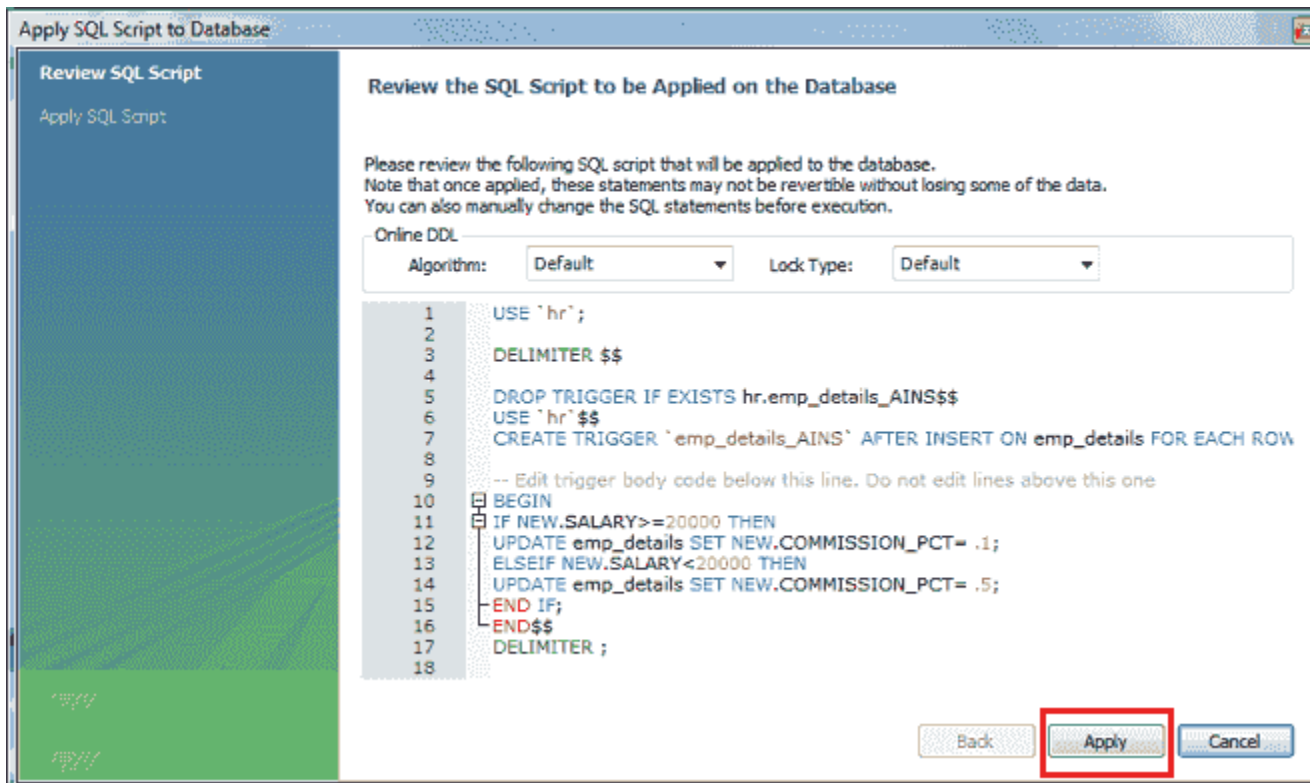
```
USE `hr`;
```

```
DELIMITER
$$
CREATE TRIGGER `emp_details_AINS`
AFTER INSERT
ON emp_details FOR EACH ROW
-- Edit trigger body code below this line. Do not edit lines above
this one
```

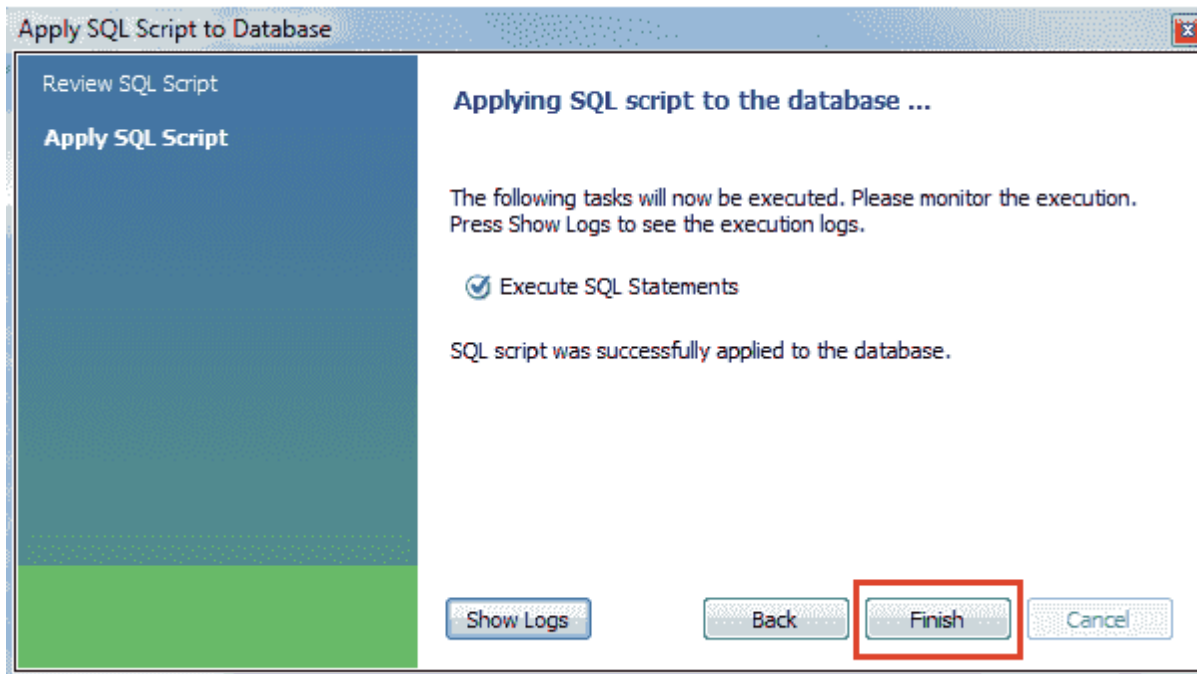
After completing the code, click on apply button.

Note : See a new text *Delete Trigger* has come in *Add Trigger* button. Clicking on this you can delete the trigger.

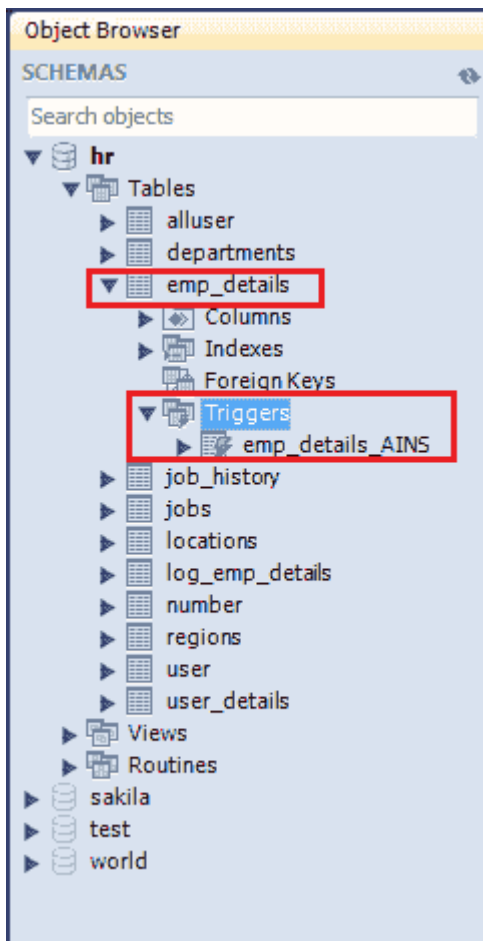
Finally you can review the script once again, as there is no error, let click on *Apply* button :



This the final window before finish. Let click on *Finish* button.



If you take a look at the schema, you will see emp_details_AINS trigger under the emp_details table as follows :



MySQL Trigger : Example AFTER INSERT

In the following example, we have two tables : `emp_details` and `log_emp_details`. To insert some information into `log_emp_details` table (which have three fields employee id and salary and edtime) every time, when an INSERT happen into `emp_details` table we have used the following trigger :

```
DELIMITER
$$
USE `hr`
$$
CREATE
DEFINER=`root`@`127.0.0.1`
TRIGGER `hr`.`emp_details_AINS`
AFTER INSERT ON `hr`.`emp_details`
FOR EACH ROW
-- Edit trigger body code below this line. Do not edit lines above
this one
BEGIN
INSERT INTO log_emp_details
VALUES(NEW.employee_id, NEW.salary, NOW());
END$$
```

Records of the table (on some columns) : **emp_details**

```
mysql> SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, JOB_ID, SALARY, C
OMMISSION_PCT FROM emp_details;
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOB_ID	SALARY	COMMISSION_PCT
100	Steven	King	AD_PRES	24000.00	0.10
101	Neena	Kochhar	AD_VP	17000.00	0.50
102	Lex	De Haan	AD_VP	17000.00	0.50
103	Alexander	Hunold	IT_PROG	9000.00	0.25
104	Bruce	Ernst	IT_PROG	6000.00	0.25

```
|          105 | David          | Austin      | IT_PROG | 4800.00 |
0.25 |
```

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

6 rows in set (0.00 sec)

Records of the table (all columns) : **log_emp_details**

```
mysql> SELECT * FROM log_emp_details;
```

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

```
| emp_details | SALARY      | EDTTIME              |
```

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

```
|          100 | 24000.00 | 2011-01-15 00:00:00 |
```

```
|          101 | 17000.00 | 2010-01-12 00:00:00 |
```

```
|          102 | 17000.00 | 2010-09-22 00:00:00 |
```

```
|          103 | 9000.00  | 2011-06-21 00:00:00 |
```


	104		6000.00		2012-07-05 00:00:00	
	105		4800.00		2011-06-21 00:00:00	

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

6 rows in set (0.02 sec)

Now insert one record in emp_details table see the records both in emp_details and log_emp_details tables :

```
mysql> INSERT INTO emp_details VALUES(236, 'RABI', 'CHANDRA', 'RABI', '590.423.45700', '2013-01-12', 'AD_VP', 15000, .5);
```

Query OK, 1 row affected (0.07 sec)

```
mysql> SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, JOB_ID, SALARY, COMMISSION_PCT FROM emp_details;
```

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

	EMPLOYEE_ID		FIRST_NAME		LAST_NAME		JOB_ID		SALARY		COMMISSION_PCT	
--	-------------	--	------------	--	-----------	--	--------	--	--------	--	----------------	--

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

	100	Steven	King	AD_PRES	24000.00	
0.10						
	101	Neena	Kochhar	AD_VP	17000.00	
0.50						
	102	Lex	De Haan	AD_VP	17000.00	
0.50						
	103	Alexander	Hunold	IT_PROG	9000.00	
0.25						
	104	Bruce	Ernst	IT_PROG	6000.00	
0.25						
	105	David	Austin	IT_PROG	4800.00	
0.25						
	236	RABI	CHANDRA	AD_VP	15000.00	
0.50						

```

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

```

7 rows in set (0.00 sec)

```
mysql> SELECT * FROM log_emp_details;
```

emp_details	SALARY	EDTTIME
100	24000.00	2011-01-15 00:00:00
101	17000.00	2010-01-12 00:00:00
102	17000.00	2010-09-22 00:00:00
103	9000.00	2011-06-21 00:00:00
104	6000.00	2012-07-05 00:00:00
105	4800.00	2011-06-21 00:00:00
236	15000.00	2013-07-15 16:52:24

7 rows in set (0.00 sec)

MySQL Trigger : Example BEFORE INSERT

In the following example, before insert a new record in emp_details table, a trigger check the column value of FIRST_NAME, LAST_NAME, JOB_ID and

- If there are any space(s) before or after the FIRST_NAME, LAST_NAME, TRIM() function will remove those.
- The value of the JOB_ID will be converted to upper cases by UPPER() function.

Here is the trigger code :

```
USE `hr`;  
DELIMITER  
$$  
CREATE TRIGGER `emp_details_BINS`  
BEFORE INSERT  
ON emp_details FOR EACH ROW  
-- Edit trigger body code below this line. Do not edit lines above  
this one  
BEGIN  
SET NEW.FIRST_NAME = TRIM(NEW.FIRST_NAME);  
SET NEW.LAST_NAME = TRIM(NEW.LAST_NAME);  
SET NEW.JOB_ID = UPPER(NEW.JOB_ID);END;  
$$
```

Now insert a row into emp_details table (check the FIRST_NAME, LAST_NAME, JOB_ID columns):

```
mysql> INSERT INTO emp_details VALUES (334, ' Ana ', ' King', 'ANA'  
, '690.432.45701', '2013-02-05', 'it_prog', 17000, .50);  
Query OK, 1 row affected (0.04 sec)
```

Now list the following fields of emp_details :

```
mysql> SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, JOB_ID, SALARY, COMMISSION_PCT FROM emp_details;
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOB_ID	SALARY	COMMISSION_PCT
100	Steven	King	AD_PRES	24000.00	0.10
101	Neena	Kochhar	AD_VP	17000.00	0.50
102	Lex	De Haan	AD_VP	17000.00	0.50
103	Alexander	Hunold	IT_PROG	9000.00	0.25
104	Bruce	Ernst	IT_PROG	6000.00	0.25
105	David	Austin	IT_PROG	4800.00	0.25

```

|          236 | RABI          | CHANDRA      | AD_VP      | 15000.00 |
0.50 |
|          334 | Ana          | King         | IT_PROG    | 17000.00 |
0.50 |
+-----+-----+-----+-----+-----+
-----+
8 rows in set (0.00 sec)

```

See the last row :

FIRST_NAME - > ' Ana ' has changed to 'Ana'

LAST_NAME - > ' King' has changed to 'King'

JOB_ID - > ' it_prog' has changed to 'IT_PROG'

MySQL Trigger : Example AFTER UPDATE

We have two tables student_mast and stu_log. student_mast have three columns

STUDENT_ID, NAME, ST_CLASS. stu_log table has two columns user_id and description.

```

mysql> SELECT * FROM STUDENT_MAST;
+-----+-----+-----+
| STUDENT_ID | NAME          | ST_CLASS |
+-----+-----+-----+
|          1 | Steven King   |          7 |
|          2 | Neena Kochhar |          8 |

```

```
|          3 | Lex  De Haan          |          8 |
|          4 | Alexander Hunold      |         10 |
+-----+-----+-----+
4 rows in set (0.00 sec)
```

Let we promote all the students in next class i.e. 7 will be 8, 8 will be 9 and so on. After updating a single row in student_mast table a new row will be inserted in stu_log table where we will store the current user id and a small description regarding the current update. Here is the trigger code :

```
-- Full Trigger
DDL Statements
-- Note: Only CREATE TRIGGER statements are allowed
DELIMITER
$$
USE `test`
$$
CREATE
DEFINER=`root`@`127.0.0.1`
TRIGGER `test`.`student_mast_AUPD`
```

```
AFTER UPDATE
ON `test`.`student_mast`FOR EACH ROW
-- Edit trigger body code below this line. Do not edit lines above
this one
BEGIN
INSERT into stu_log VALUES (user(), CONCAT('Update Student Record '
,
      OLD.NAME,' Previous Class :',OLD.ST_CLASS,' Present Class
',
      NEW.st_class));
END
$$
```

After update STUDENT_MAST table :

```
mysql> UPDATE STUDENT_MAST SET ST_CLASS = ST_CLASS + 1;
Query OK, 4 rows affected (0.20 sec)
Rows matched: 4
Changed: 4
Warnings: 0
```


The trigger show you the updated records in 'stu_log'. Here is the latest position of STUDENT_MAST and STU_LOG tables :

```
mysql> SELECT * FROM STUDENT_MAST;
```

STUDENT_ID	NAME	ST_CLASS
1	Steven King	8
2	Neena Kochhar	9
3	Lex De Haan	9
4	Alexander Hunold	11

```
4 rows in set (0.00 sec)mysql> SELECT * FROM STU_LOG;
```

user_id	description
---------	-------------

```
| root@localhost | Update Student Record Steven King Previous Class
:7 Present Class 8 |
| root@localhost | Update Student Record Neena Kochhar Previous Cl
ass :8 Present Class 9 |
| root@localhost | Update Student Record Lex De Haan Previous Clas
s :8 Present Class 9 |
| root@localhost | Update Student Record Alexander Hunold Previous
Class :10 Present Class 11|
+-----+-----+-----+-----+-----+-----+-----+-----+
-----+
4 rows in set (0.00 sec)
```

MySQL Trigger : Example BEFORE UPDATE

We have a table student_marks with 10 columns and 4 rows. There are data only in STUDENT_ID and NAME columns.

```
mysql> SELECT * FROM STUDENT_MARKS;
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+
| STUDENT_ID | NAME | SUB1 | SUB2 | SUB3 | SUB4 | SUB5 |
| TOTAL | PER_MARKS | GRADE |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
```

1	Steven King	0	0	0	0	0
0	0.00					
2	Neena Kochhar	0	0	0	0	0
0	0.00					
3	Lex De Haan	0	0	0	0	0
0	0.00					
4	Alexander Hunold	0	0	0	0	0
0	0.00					

4 rows in set (0.00 sec)

Now the exam is over and we have received all subject marks, now we will update the table, total marks of all subject, the percentage of total marks and grade will be automatically calculated. For this sample calculation, the following conditions are assumed :

Total Marks (will be stored in TOTAL column) : $TOTAL = SUB1 + SUB2 + SUB3 + SUB4 + SUB5$

Percentage of Marks (will be stored in PER_MARKS column) : $PER_MARKS = (TOTAL)/5$

Grade (will be stored GRADE column) :

- If $PER_MARKS \geq 90 \rightarrow 'EXCELLENT'$

- If PER_MARKS >= 75 AND PER_MARKS < 90 -> 'VERY GOOD'
- If PER_MARKS >= 60 AND PER_MARKS < 75 -> 'GOOD'
- If PER_MARKS >= 40 AND PER_MARKS < 60 -> 'AVERAGE'
- If PER_MARKS < 40 -> 'NOT PROMOTED'

Here is the code :

```
mysql> UPDATE STUDENT_MARKS SET SUB1 = 54, SUB2 = 69, SUB3 = 89, SUB4 = 87, SUB5 = 59 WHERE STUDENT_ID = 1;
```

Query OK, 1 row affected (0.05 sec)

Rows matched: 1

Changed: 1

Warnings: 0

Let update the marks of a student :

```
USE `test`;
```

```
DELIMITER
```

```
$$
```

```
CREATE TRIGGER `student_marks_BUPD`
```

```
BEFORE UPDATE
```

```
ON student_marks FOR EACH ROW
```

```
-- Edit trigger body code below this line. Do not edit lines above
this one

BEGIN

SET NEW.TOTAL = NEW.SUB1 + NEW.SUB2 + NEW.SUB3 + NEW.SUB4 + NEW.SUB
5;

SET NEW.PER_MARKS = NEW.TOTAL/5;

IF NEW.PER_MARKS >=90 THEN
SET NEW.GRADE = 'EXCELLENT';
ELSEIF NEW.PER_MARKS>=75 AND NEW.PER_MARKS<90 THEN
SET NEW.GRADE = 'VERY GOOD';
ELSEIF NEW.PER_MARKS>=60 AND NEW.PER_MARKS<75 THEN
SET NEW.GRADE = 'GOOD';
ELSEIF NEW.PER_MARKS>=40 AND NEW.PER_MARKS<60 THEN
SET NEW.GRADE = 'AVERAGE';
ELSESET NEW.GRADE = 'NOT PROMOTED';
END IF;

END;

$$
```

Now check the STUDENT_MARKS table with updated data. The trigger show you the updated records in 'stu_log'.

```
mysql> SELECT * FROM STUDENT_MARKS;
```

STUDENT_ID	NAME	SUB1	SUB2	SUB3	SUB4	SUB5
TOTAL	PER_MARKS	GRADE				
1	Steven King	54	69	89	87	59
358	71.60	GOOD				
2	Neena Kochhar	0	0	0	0	0
0	0.00					
3	Lex De Haan	0	0	0	0	0
0	0.00					
4	Alexander Hunold	0	0	0	0	0
0	0.00					

4 rows in set (0.00 sec)

MySQL Trigger : Example AFTER DELETE

In our 'AFTER UPDATE' example, we had two tables student_mast and stu_log. student_mast have three columns STUDENT_ID, NAME, ST_CLASS and stu_log table has two columns user_id and description. We want to store some information in stu_log table after a delete operation happened on student_mast table. Here is the trigger :

```
USE `test`;  
  
DELIMITER  
  
$$  
  
CREATE TRIGGER `student_mast_ADEL`  
AFTER DELETE ON student_mast FOR EACH ROW  
-- Edit trigger body code below this line. Do not edit lines above  
this one  
BEGIN  
INSERT into stu_log VALUES (user(), CONCAT('Update Student Record '  
,  
      OLD.NAME, ' Clas :', OLD.ST_CLASS, '-> Deleted on ', NOW()))  
;  
END;  
  
$$
```

Let delete a student from STUDENT_MAST.

```
mysql> DELETE FROM STUDENT_MAST WHERE STUDENT_ID = 1;
```

```
Query OK, 1 row affected (0.06 sec)
```

Here is the latest position of STUDENT_MAST, STU_LOG tables :

```
mysql> SELECT * FROM STUDENT_MAST;
```

STUDENT_ID	NAME	ST_CLASS
2	Neena Kochhar	9
3	Lex De Haan	9
4	Alexander Hunold	11

```
3 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM STU_LOG;
```

user_id	description
---------	-------------


```

+-----+-----+
-----+
| root@localhost | Update Student RecordSteven King Previous Class
:7 Present Class 8          |
| root@localhost | Update Student RecordNeena Kochhar Previous Cla
ss :8 Present Class 9          |
| root@localhost | Update Student RecordLex De Haan Previous Class
:8 Present Class 9          |
| root@localhost | Update Student RecordAlexander Hunold Previous C
lass :10 Present Class 11      |
| root@localhost | Update Student Record Steven King Clas :8-> Dele
ted on 2013-07-16 15:35:30    |
+-----+-----+
-----+

5 rows in set (0.00 sec)

```

How MySQL handle errors during trigger execution?

- If a BEFORE trigger fails, the operation on the corresponding row is not performed.
- A BEFORE trigger is activated by the attempt to insert or modify the row, regardless of whether the attempt subsequently succeeds.

- An AFTER trigger is executed only if any BEFORE triggers and the row operation execute successfully.
- An error during either a BEFORE or AFTER trigger results in failure of the entire statement that caused trigger invocation.
- For transactional tables, failure of a statement should cause a rollback of all changes performed by the statement.

Delete a MySQL trigger

To delete or destroy a trigger, use a DROP TRIGGER statement. You must specify the schema name if the trigger is not in the default (current) schema :

```
DROP TRIGGER [IF EXISTS] [schema_name.]trigger_name
```

if you drop a table, any triggers for the table are also dropped.

-- Partition high level

What is Partitioning?

Partitioning (a database design technique) improves performance, manageability, simplifies maintenance and reduce the cost of storing large amounts of data. Partitioning can be achieved without splitting tables by physically putting tables on individual disk drives. Partitioning allows tables,

indexes, and index-organized tables to be subdivided into smaller pieces, therefore queries that access only a fraction of the data can run faster because there is fewer data to scan. There are two major forms of partitioning :

- **Horizontal Partitioning** : Horizontal partitioning divides table rows into multiple partitions (based on a logic). All columns defined to a table are found in each partition, so no actual table attributes are missing. All the partition can be addressed individually or collectively. For example, a table that contains whole year sale transaction being partitioned horizontally into twelve distinct partitions, where each partition contains one month's data.
- **Vertical Partitioning** : Vertical partitioning divides a table into multiple tables that contain fewer columns. Like horizontal partitioning, in vertical partitioning a query scan fewer data which increases query performance. For example, a table that contains a number of very wide text or BLOB columns that aren't addressed often being broken into two tables that have the most referenced columns in one table and the text or BLOB data in another.

MySQL partitioning

Version : MySQL 5.6

MySQL supports basic table partitioning but does not support vertical partitioning (MySQL 5.6). This section describes in detail how to implement partitioning as part of your database.

How to partition a table?

In MySQL you can partition a table using CREATE TABLE or ALTER TABLE command. See the following CREATE TABLE syntax :

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
```

```
(create_definition,...)
```

```
[table_options]
```

```
[partition_options]
```

partition_options :

```
PARTITION BY
```

```
{ [LINEAR] HASH(expr)
```

```
| [LINEAR] KEY(column_list)
```

```
| RANGE(expr)
```

```
| LIST(expr) }
```

```
[PARTITIONS num]
```

```
[SUBPARTITION BY
```

```
        { [LINEAR] HASH(expr)

        | [LINEAR] KEY(column_list) }

    [SUBPARTITIONS num]

]

[(partition_definition [, partition_definition] ...)
```

partition_definition :

```
PARTITION partition_name

    [VALUES

        {LESS THAN {(expr) | MAXVALUE}

        |

        IN (value_list)}]

    [[STORAGE] ENGINE [=] engine_name]
```

```
[COMMENT [=] 'comment_text' ]
```

```
[DATA DIRECTORY [=] 'data_dir']
```

```
[INDEX DIRECTORY [=] 'index_dir']
```

```
[MAX_ROWS [=] max_number_of_rows]
```

```
[MIN_ROWS [=] min_number_of_rows]
```

```
[TABLESPACE [=] tablespace_name]
```

```
[NODEGROUP [=] node_group_id]
```

```
[(subpartition_definition [, subpartition_definition] ...)]
```

subpartition_definition :

```
SUBPARTITION logical_name
```

```
[[STORAGE] ENGINE [=] engine_name]
```

```
[COMMENT [=] 'comment_text' ]
```

```
[DATA DIRECTORY [=] 'data_dir']
```

```
[INDEX DIRECTORY [=] 'index_dir']
```

```
[MAX_ROWS [=] max_number_of_rows]
```

```
[MIN_ROWS [=] min_number_of_rows]
```

```
[TABLESPACE [=] tablespace_name]
```

```
[NODEGROUP [=] node_group_id]
```

ALTER TABLE : Partition operations

ALTER TABLE statement can be used for adding, dropping, merging, and splitting partitions, and for performing partitioning maintenance. Here we have defined a nonpartitioned table :

```
CREATE TABLE sale_mast (
```

```
    bill_no INT,
```

```
bill_date DATETIME  
  
);
```

This table can be partitioned by HASH (or in another type), using the bill_no column as the partitioning key, into 6 (or other) partitions using ALTER TABLE statement :

```
ALTER TABLE t1  
  
PARTITION BY HASH(id)  
  
PARTITIONS 6;
```

Partition naming :

Names of partitions follow the rules of other [MySQL identifiers](#) such as databases, tables, constraint, stored procedure etc. Partition names are not case-sensitive.

Advantages of partitioning

- During the scan operation, MySQL optimizer accesses those partitions that will satisfy a particular query. For example, a whole year sale records table may be broken up into 4 partitions (i.e. sale data from of Apr-Jun (partition p0), Jul-Sep (partition p1) , Oct-Dec (partition p2), Jan-Mar (partition p0)) . If a query is issued that contains sale data between

Jul-Sep quarter, then it scans the partition p1 only instead of total table records and the query will complete much sooner.

- Partitioning allows you to have more control over how data is managed inside the database. For example, you can drop specific partitions in a partitioned table where data loses its usefulness. The process of adding new data, in some cases, be greatly facilitated by adding one or more new partitions for storing that data using ALTER TABLE command.
- In partitioning, it is possible to store more data in one table than can be held on a single disk or file system partition.
- MySQL 5.6 supports explicit partition selection for queries. For example, `SELECT * FROM table1 PARTITION (p0,p1) WHERE col1 < 10` selects only those rows in partitions p0 and p1 that match the WHERE condition, this can greatly speed up queries
- Partition selection also supports the data modification statements DELETE, INSERT, REPLACE, UPDATE, and LOAD DATA, LOAD XML.

Types of MySQL partitioning

Following types of partitioning are available in MySQL 5.6 :

- [RANGE Partitioning](#)
- [LIST Partitioning](#)
- [COLUMNS Partitioning](#)
- [HASH Partitioning](#)
- [KEY Partitioning](#)
- [Subpartitioning](#)

MySQL RANGE Partitioning

In MySQL, RANGE partitioning mode allows us to specify various ranges for which data is assigned. Ranges should be contiguous but not overlapping, and are defined using the VALUES LESS THAN operator. In the following example, sale_mast table contains four columns bill_no, bill_date, cust_code and amount. This table can be partitioned by range in various of ways, depending on your requirement. Here we have used the bill_date column and decide to partition the table 4 ways by adding a PARTITION BY RANGE clause. In these partitions the range of the sale date (sale_date) are as of follow :

- partition p0 (sale between 01-01-2013 to 31-03-2013)
- partition p1 (sale between 01-04-2013 to 30-06-2013)
- partition p2 (sale between 01-07-2013 to 30-09-2013)
- partition p3 (sale between 01-10-2013 to 30-12-2013)

Let create the table :

```
mysql> CREATE TABLE sale_mast (bill_no INT NOT NULL, bill_date TIMESTAMP NOT NULL,  
  
cust_code VARCHAR(15) NOT NULL, amount DECIMAL(8,2) NOT NULL)  
  
PARTITION BY RANGE (UNIX_TIMESTAMP(bill_date)) (
```

```
PARTITION p0 VALUES LESS THAN (UNIX_TIMESTAMP('2013-04-01')),  
  
PARTITION p1 VALUES LESS THAN (UNIX_TIMESTAMP('2013-07-01')),  
  
PARTITION p2 VALUES LESS THAN (UNIX_TIMESTAMP('2013-10-01')),  
  
PARTITION p3 VALUES LESS THAN (UNIX_TIMESTAMP('2014-01-01')));
```

Query OK, 0 rows affected (1.50 sec)

Now insert some records in sale_mast table :

```
mysql> INSERT INTO sale_mast VALUES (1, '2013-01-02', 'C001', 125.56),  
  
(2, '2013-01-25', 'C003', 456.50),  
  
(3, '2013-02-15', 'C012', 365.00),  
  
(4, '2013-03-26', 'C345', 785.00),  
  
(5, '2013-04-19', 'C234', 656.00),  
  
(6, '2013-05-31', 'C743', 854.00),
```

```
(7, '2013-06-11', 'C234', 542.00),
```

```
(8, '2013-07-24', 'C003', 300.00),
```

```
(8, '2013-08-02', 'C456', 475.20);
```

Query OK, 9 rows affected (0.07 sec)

Records: 9 Duplicates: 0 Warnings: 0

```
mysql> SELECT * FROM sale_mast;
```

```
+-----+-----+-----+-----+
| bill_no | bill_date           | cust_code | amount |
+-----+-----+-----+-----+
|      1 | 2013-01-02 00:00:00 | C001      | 125.56 |
```

```
|      2 | 2013-01-25 00:00:00 | C003      | 456.50 |
|      3 | 2013-02-15 00:00:00 | C012      | 365.00 |
|      4 | 2013-03-26 00:00:00 | C345      | 785.00 |
|      5 | 2013-04-19 00:00:00 | C234      | 656.00 |
|      6 | 2013-05-31 00:00:00 | C743      | 854.00 |
|      7 | 2013-06-11 00:00:00 | C234      | 542.00 |
|      8 | 2013-07-24 00:00:00 | C003      | 300.00 |
|      9 | 2013-08-02 00:00:00 | C456      | 475.20 |
```

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

```
9 rows in set (0.00 sec)
```

Here is the partition status of sale_mast table :

```
mysql> SELECT PARTITION_NAME, TABLE_ROWS FROM INFORMATION_SCHEMA.PARTITIONS WHERE T  
ABLE_NAME='sale_mast';
```

```
+-----+-----+  
  
| PARTITION_NAME | TABLE_ROWS |  
  
+-----+-----+  
  
| p0           |          4 |  
  
| p1           |          3 |  
  
| p2           |          2 |  
  
| p3           |          0 |  
  
+-----+-----+
```

```
4 rows in set (0.02 sec)
```

In the above way you can partition the table based on sale amount (amount). In these partitions the range of the sale amount (amount) are as of follow :

- partition p0 (sale amount < 100)
- partition p1 (sale amount < 500)
- partition p2 (sale amount <1000)
- partition p3 (sale amount<1500)

Let create the table :

```
mysql> CREATE TABLE sale_mast1 (bill_no INT NOT NULL, bill_date TIMESTAMP NOT NULL,  
  
cust_code VARCHAR(15) NOT NULL, amount INT NOT NULL)  
  
PARTITION BY RANGE (amount) (  
  
PARTITION p0 VALUES LESS THAN (100),  
  
PARTITION p1 VALUES LESS THAN (500),  
  
PARTITION p2 VALUES LESS THAN (1000),
```

```
PARTITION p3 VALUES LESS THAN (1500));
```

```
Query OK, 0 rows affected (1.34 sec)
```

Drop a MySQL partition

If you feel some data are useless in a partitioned table you can drop one or more partition(s). To delete all rows from partition p0 of sale_mast, you can use the following statement :

```
MySQL> ALTER TABLE sale_mast TRUNCATE PARTITION p0;
```

```
Query OK, 0 rows affected (0.49 sec)
```

```
mysql> SELECT * FROM sale_mast;
```

```
+-----+-----+-----+-----+
| bill_no | bill_date          | cust_code | amount |
+-----+-----+-----+-----+
```


	5		2013-04-19 00:00:00		C234		656.00	
	6		2013-05-31 00:00:00		C743		854.00	
	7		2013-06-11 00:00:00		C234		542.00	
	8		2013-07-24 00:00:00		C003		300.00	
	9		2013-08-02 00:00:00		C456		475.20	

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

5 rows in set (0.01 sec)

Here is the partition status of sale_mast after dropping the partition p0 :

```
MySQL> SELECT PARTITION_NAME, TABLE_ROWS FROM INFORMATION_SCHEMA.PARTITIONS
WHERE TABLE_NAME='sale_mast';
```

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

	PARTITION_NAME		TABLE_ROWS	
--	----------------	--	------------	--

```

+-----+-----+
| p0          |          0 |
| p1          |          3 |
| p2          |          2 |
| p3          |          0 |
+-----+-----+
4 rows in set (0.05 sec)

```

MySQL LIST Partitioning

List partition allows us to segment data based on a pre-defined set of values (e.g. 1, 2, 3). This is done by using `PARTITION BY LIST(expr)` where `expr` is a column value and then defining each partition by means of a `VALUES IN (value_list)`, where `value_list` is a comma-separated list of integers. In MySQL 5.6, it is possible to match against only a list of integers (and possibly NULL) when partitioning by LIST. In the following example, `sale_mast2` table contains four columns `bill_no`, `bill_date`, `agent_code`, and `amount`. Suppose there are 11 agents represent three cities A, B, C these can be arranged in three partitions with LIST Partitioning as follows :

City	Agent ID
------	----------

A	1, 2, 3
B	4, 5, 6
C	7, 8, 9, 10, 11

Let create the table :

```
mysql> CREATE TABLE sale_mast2 (bill_no INT NOT NULL, bill_date TIMESTAMP NOT NULL,  
  
agent_code INT NOT NULL, amount INT NOT NULL)  
  
PARTITION BY LIST(agent_code) (  
  
PARTITION pA VALUES IN (1,2,3),  
  
PARTITION pB VALUES IN (4,5,6),  
  
PARTITION pC VALUES IN (7,8,9,10,11));  
  
Query OK, 0 rows affected (1.17 sec)
```

MySQL COLUMNS Partitioning

In COLUMNS partitioning it is possible to use multiple columns in partitioning keys. There are two types of COLUMNS partitioning :

- [RANGE COLUMNS partitioning](#)
- [LIST COLUMNS partitioning](#)

In addition, both RANGE COLUMNS partitioning and LIST COLUMNS partitioning support the use of non-integer columns for defining value ranges or list members. The permitted data types are shown in the following list:

Both RANGE COLUMNS partitioning and LIST COLUMNS partitioning support following data types for defining value ranges or list members.

- All integer types: TINYINT, SMALLINT, MEDIUMINT, INT (INTEGER), and BIGINT.
- DATE and DATETIME.

RANGE COLUMNS partitioning

RANGE COLUMNS partitioning is similar to range partitioning with some significant difference. RANGE COLUMNS accepts a list of one or more columns as partition keys. You can define the ranges using various columns of types (mentioned above) other than integer types.

Here is the basic syntax for creating a table partitioned by RANGE COLUMNS :

```
CREATE TABLE table_name

PARTITIONED BY RANGE COLUMNS(column_list) (

    PARTITION partition_name VALUES LESS THAN (value_list)[,

    PARTITION partition_name VALUES LESS THAN (value_list)][,

    ...]

)
```

column_list:

```
column_name[, column_name][, ...]
```

value_list:

```
value[, value][, ...]
```

- column_list is a list of one or more columns.
- value_list is a list of values and must be supplied for each partition definition.
- column list and in the value list defining each partition must occur in the same order
- The order of the column names in the partitioning column list and the value lists do not have to be the same as the order of the table column definitions in CREATE TABLE statement.

Here is an example :

```
mysql> CREATE TABLE table3 (col1 INT, col2 INT, col3 CHAR(5), col4 INT)

PARTITION BY RANGE COLUMNS(col1, col2, col3)

(PARTITION p0 VALUES LESS THAN (50, 100, 'aaaaa'),

PARTITION p1 VALUES LESS THAN (100,200,'bbbbbb'),

PARTITION p2 VALUES LESS THAN (150,300,'ccccc'),

PARTITION p3 VALUES LESS THAN (MAXVALUE, MAXVALUE, MAXVALUE));
```

```
Query OK, 0 rows affected (1.39 sec)
```

In the above example -

- Table table3 contains the columns col1, col2, col3, col4
- The first three columns have participated in partitioning COLUMNS clause, in the order col1, col2, col3.
- Each value list used to define a partition contains 3 values in the same order and (INT, INT, CHAR(5)) form.

LIST COLUMNS partitioning

LIST COLUMNS accepts a list of one or more columns as partition keys. You can use various columns of data of types other than integer types as partitioning columns. You can use string types, DATE, and DATETIME columns

In a company there are agents in 3 cities, for sales and marketing purposes. We have organized the agents in 3 cities as shown in the following table :

City	Agent ID
A	A1, A2, A3

B	B1, B2, B3
C	C1, C2, C3, C4, C5

Let create a table with LIST COLUMNS partitioning based on the above information :

```
mysql> CREATE TABLE salemast ( agent_id VARCHAR(15), agent_name VARCHAR(50),
agent_address VARCHAR(100), city_code VARCHAR(10))
PARTITION BY LIST COLUMNS(agent_id) (
PARTITION pcity_a VALUES IN('A1', 'A2', 'A3'),
PARTITION pcity_b VALUES IN('B1', 'B2', 'B3'),
PARTITION pcity_c VALUES IN ('C1', 'C2', 'C3', 'C4', 'C5'));
Query OK, 0 rows affected (1.06 sec)
```


You can use DATE and DATETIME columns in LIST COLUMNS partitioning, see the following example :

```
CREATE TABLE sale_master (bill_no INT NOT NULL, bill_date DATE,  
  
cust_code VARCHAR(15) NOT NULL, amount DECIMAL(8,2) NOT NULL)  
  
PARTITION BY RANGE COLUMNS (bill_date) (  
  
PARTITION p_qtr1 VALUES LESS THAN ('2013-04-01'),  
  
PARTITION p_qtr2 VALUES LESS THAN ('2013-07-01'),  
  
PARTITION p_qtr3 VALUES LESS THAN ('2013-10-01'),  
  
PARTITION p_qtr4 VALUES LESS THAN ('2014-01-01'));
```

MySQL HASH Partitioning

MySQL HASH partition is used to distribute data among a predefined number of partitions on a column value or expression based on a column value. This is done by using PARTITION BY HASH(expr) clause, adding in CREATE TABLE STATEMENT. In PARTITIONS num clause, num is a positive integer represents the number of partitions of the table. The following statement

creates a table that uses hashing on the student_id column and is divided into 4 partitions :

```
MySQL>CREATE TABLE student (student_id INT NOT NULL,  
  
class VARCHAR(8), name VARCHAR(40),  
  
date_of_admission DATE NOT NULL DEFAULT '2000-01-01')  
  
PARTITION BY HASH(student_id)  
  
PARTITIONS 4;  
  
Query OK, 0 rows affected (1.43 sec)
```

It is also possible to make a partition based on the year in which a student was admitted. See the following statement :

```
MySQL> CREATE TABLE student (student_id INT NOT NULL,  
  
class VARCHAR(8), class VARCHAR(8), name VARCHAR(40),  
  
date_of_admission DATE NOT NULL DEFAULT '2000-01-01')
```

```
PARTITION BY HASH(YEAR(date_of_admission))
```

```
PARTITIONS 4;
```

```
Query OK, 0 rows affected (1.27 sec)
```

MySQL KEY Partitioning

MySQL KEY partition is a special form of HASH partition, where the hashing function for key partitioning is supplied by the MySQL server. The server employs its own internal hashing function which is based on the same algorithm as PASSWORD(). This is done by using PARTITION BY KEY, adding in CREATE TABLE STATEMENT. In KEY partitioning KEY takes only a list of zero or more column names. Any columns used as the partitioning key must comprise part or all of the table's primary key if the table has one. If there is a primary key in a table, it is used as partitioning key when no column is specified as the partitioning key. Here is an example :

```
MySQL> CREATE TABLE table1 ( id INT NOT NULL PRIMARY KEY,
```

```
fname VARCHAR(25), lname VARCHAR(25))
```

```
PARTITION BY KEY()
```

```
PARTITIONS 2;
```

```
Query OK, 0 rows affected (0.84 sec)
```

If there is no primary key but there is a unique key in a table, then the unique key is used for the partitioning key :

```
MySQL> CREATE TABLE table2 ( id INT NOT NULL, fname VARCHAR(25),
```

```
lname VARCHAR(25),
```

```
UNIQUE KEY (id))
```

```
PARTITION BY KEY()
```

```
PARTITIONS 2;
```

```
Query OK, 0 rows affected (0.77 sec)
```

MySQL Subpartitioning

Subpartitioning is a method to divide each partition further in a partitioned table. See the following CREATE TABLE statement :

```
CREATE TABLE table10 (BILL_NO INT, sale_date DATE, cust_code VARCHAR(15),  
  
AMOUNT DECIMAL(8,2))  
  
    PARTITION BY RANGE(YEAR(sale_date) )  
  
    SUBPARTITION BY HASH(TO_DAYS(sale_date))  
  
    SUBPARTITIONS 4 (  
  
        PARTITION p0 VALUES LESS THAN (1990),  
  
        PARTITION p1 VALUES LESS THAN (2000),  
  
        PARTITION p2 VALUES LESS THAN (2010),  
  
        PARTITION p3 VALUES LESS THAN MAXVALUE  
  
    );
```

In the above statement -

- The table has 4 RANGE partitions.

- Each of these partitions—p0, p1, p2 and p3—is further divided into 4 subpartitions.
- Therefore the entire table is divided into $4 * 4 = 16$ partitions.

Here is the partition status of table10 :

```
mysql> SELECT PARTITION_NAME, TABLE_ROWS FROM  
INFORMATION_SCHEMA.PARTITIONS WHERE TABLE_NAME='stable';
```

```
+-----+-----+  
| PARTITION_NAME | TABLE_ROWS |  
+-----+-----+  
| p0             | 0           |  
| p0             | 0           |  
| p0             | 0           |  
| p0             | 0           |
```

p1		0	
----	--	---	--

p1		0	
----	--	---	--

p1		0	
----	--	---	--

p1		0	
----	--	---	--

p2		0	
----	--	---	--

p2		0	
----	--	---	--

p2		0	
----	--	---	--

p2		0	
----	--	---	--

p3		0	
----	--	---	--

p3		0	
----	--	---	--

p3		0	
----	--	---	--

```
| p3 | 0 |
```

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

```
16 rows in set (0.16 sec)
```

---- NOW WHAT IS LEFT

normalization table splitting techniques

er diagram 1:many etc

procedure function

trigger