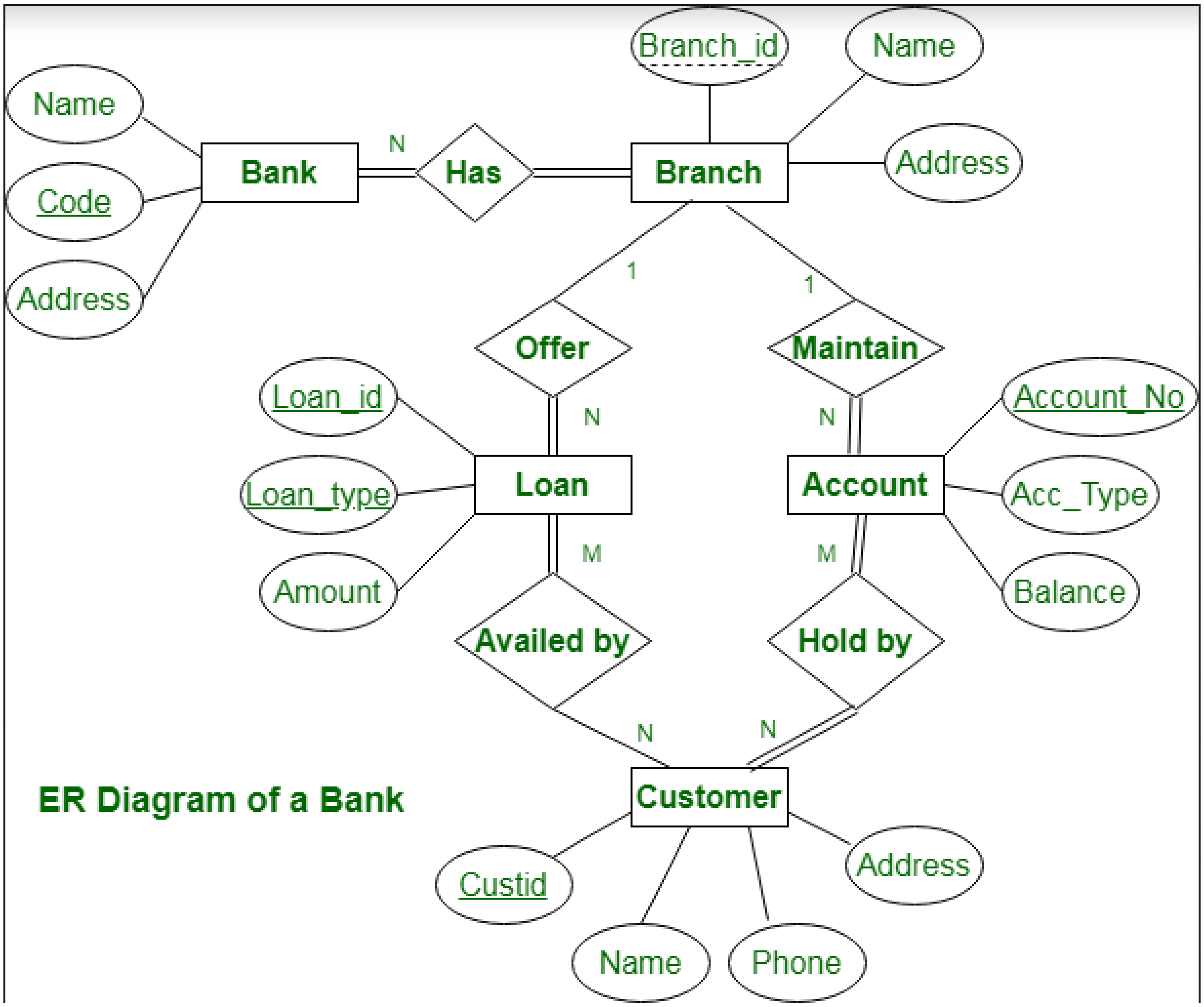
ER diagram of Bank Management System

ER diagram is known as Entity-Relationship diagram. It is used to analyze to structure of the Database. It shows relationships between entities and their attributes. An ER model provides a means of communication.

ER diagram of Bank has the following description :

- Bank have Customer.
- Banks are identified by a name, code, address of main office.
- Banks have branches.
- Branches are identified by a branch_no., branch_name, address.
- Customers are identified by name, cust-id, phone number, address.
- Customer can have one or more accounts.
- Accounts are identified by account_no., acc_type, balance.
- Customer can avail loans.
- Loans are identified by loan_id, loan_type and amount.
- Account and loans are related to bank’s branch.

ER Diagram of Bank Management System :



This bank ER diagram illustrates key information about bank, including entities such as branches, customers, accounts, and loans. It allows us to understand the relationships between entities.

Entities and their **Attributes** are :

- **Bank Entity :** Attributes of Bank Entity are Bank Name, Code and Address.
  Code is Primary Key for Bank Entity.
- **Customer Entity :** Attributes of Customer Entity are Customer_id, Name, Phone Number and Address.
  Customer_id is Primary Key for Customer Entity.
- **Branch Entity :** Attributes of Branch Entity are Branch_id, Name and Address.
  Branch_id is Primary Key for Branch Entity.
- **Account Entity :** Attributes of Account Entity are Account_number, Account_Type and Balance.
  Account_number is Primary Key for Account Entity.
- **Loan Entity :** Attributes of Loan Entity are Loan_id, Loan_Type and Amount.

Loan_id is Primary Key for Loan Entity.

ER Diagram for E-Commerce Website

E-Commerce Website allows easy management of **products**, **orders**, **users**, **addresses**, **payments**, **tracking** information as well as **cart management**. **Products** are categorized in a user-friendly manner, as well as users can register and create multiple addresses in one account. **Orders** are placed by users.

Payment methods are diversified which is aimed at enhancing **security** and **flexibility** in **transactions**. **Tracking details** allow users to access their Order status from processing through to delivery. The **shopping cart** management system enables users to add, and check out products thereby helping with a smooth shopping process.

Entities and Attributes for the E-commerce Website

Entities and Attributes are defined below:

1. Product:

- **P-ID(Primary Key):** Unique identifier for each product.
- **Name:** Name of the product.
- **Price:** Price of the product.
- **Description:** Description of the product.

2. Pro-category:

- **Category – ID(Primary Key):** Unique identifier for each category.
- **Name:** Name of the category.

3. Order:

- **Order – No(Primary Key):** Unique identifier for each order.
- **Order – Amount:**
- **Order – Date:**

4. User:

- **User – ID(Primary Key):** Unique identifier for each user or customer.
- **Name:** Name of the user.
- **Email:** Email of the user.

5. Address:

- **Address – ID(Primary Key):** Unique identifier for each address
- **Country:** Country of the user.
- **State:** State of the user.
- **City:** City of the user.
- **Pin – code:** Pin code of the user.

6. Payment:

- **Payment – ID(Primary Key):** Unique identifier for each payment.

- **Method:** Payment method like UPI or Credit Card etc.
- **Amount:** Total amount paid by user.

7. Tracking Detail:

- **Tracking – ID(Primary Key):** Unique identifier for each tracking detail.
- **Status:** Tracking status like on the way or delivered etc.
- **Order – No(Foreign Key):** Reference to the order which need to be tracked.

8. Cart:

- **Cart – ID(Primary Key):** Unique identifier for each cart.
- **User – ID(Foreign Key):** Reference to the user.

Relationships Between These Entities

1. Product – Prod-Category Relationship:

- One product can belong to only one category.
- One category can have multiple products.
- So this is a **Many-to-one** relationship, showing that many products can belong to a single category.

2. Order-User Relationship:

- One user can place multiple orders.
- Each order is placed by exactly one user.
- This is a **one-to-many** relationship, showing that a user can place multiple orders, but each order is placed by exactly one user.

3. User-Address Relationship:

- One user can have multiple addresses.
- Each address belongs to exactly one user.
- This is a **one-to-many** relationship, indicating that a user can have multiple addresses associated with their account.

4. Tracking Detail – Order Relationship:

- One order can have multiple tracking details.
- Each tracking detail corresponds to exactly one order.
- This is **Many-to-one** relationships showing that an order can have one or multiple associated tracking details.

5. Product – Cart Relationship:

- One product can be added to multiple carts.
- Each cart can contain multiple products.
- This is **many-to-many** relationship.

6. User-Payment Relationship:

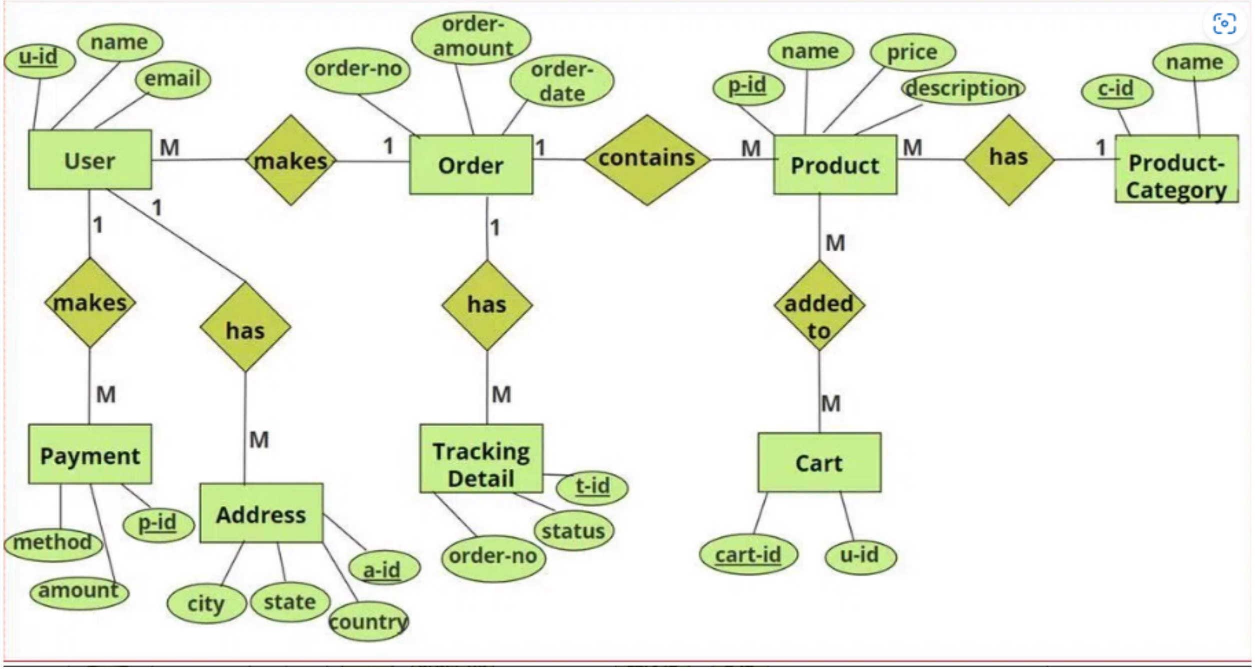
- One user can make multiple payments.
- Each payment is made by exactly one user.
- This is **one-to-many** relationship because each user can make multiple payments, and each payment is made by a single user.

7. Order-Product Relationship:

- One order can contains multiple products.
- Many products are get ordered in each order.
- So this is **one-to-many** relationship we can order multiple products on each order.

Representation of ER Diagram

Below is the ER diagram of large scale E-commerce platforms which meets all our requirements:


u-id
name
email
User
M
makes
1
order-no
order-amount
order-date
Order
1
contains
M
p-id
name
price
description
Product
M
has
1
c-id
name
Product-Category
1
1
makes
M
Payment
method
amount
p-id
has
M
Address
city
state
country
a-id
1
has
M
Tracking Detail
t-id
status
order-no
M
added to
M
Cart
cart-id
u-id

Normalization

- Normalization is the process of organizing the data in the database.

- Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate undesirable characteristics like Insertion, Update, and Deletion Anomalies.

- Normalization divides the larger table into smaller and links them using relationships.

- The normal form is used to reduce redundancy from the database table.

Why do we need Normalization?

The main reason for normalizing the relations is removing these anomalies. Failure to eliminate anomalies leads to data redundancy and can cause data integrity and other problems as the database grows. Normalization consists of a series of guidelines that helps to guide you in creating a good database structure.

Data modification anomalies can be categorized into three types:

- **Insertion Anomaly:** Insertion Anomaly refers to when one cannot insert a new tuple into a relationship due to lack of data.

- **Deletion Anomaly:** The delete anomaly refers to the situation where the deletion of data results in the unintended loss of some other important data.

- **Updatation Anomaly:** The update anomaly is when an update of a single data value requires multiple rows of data to be updated.

Normal Form	Description
1NF	A relation is in 1NF if it contains an atomic value.
2NF	A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key.
3NF	A relation will be in 3NF if it is in 2NF and no transition dependency exists.
BCNF	A stronger definition of 3NF is known as Boyce Codd's normal form.
4NF	A relation will be in 4NF if it is in Boyce Codd's normal form and has no multi-valued dependency.
5NF	A relation is in 5NF. If it is in 4NF and does not

contain any join dependency, joining should be lossless.

Advantages of Normalization

- Normalization helps to minimize data redundancy.
- Greater overall database organization.
- Data consistency within the database.

Disadvantages of Normalization

- You cannot start building the database before knowing what the user needs.
- The performance degrades when normalizing the relations to higher normal forms, i.e., 4NF, 5NF.
- It is very time-consuming and difficult to normalize relations of a higher degree.

First Normal Form (1NF)

- A relation will be 1NF if it contains an atomic value.
- It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attribute.
- First normal form disallows the multi-valued attribute, composite attribute, and their combinations.

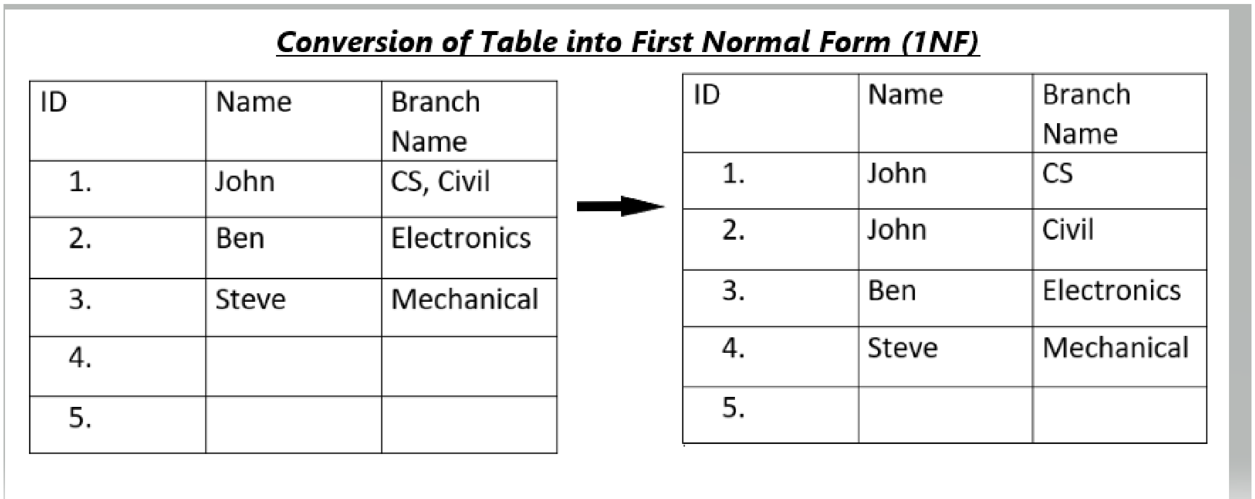
Example: Relation EMPLOYEE is not in 1NF because of multi-valued attribute EMP_PHONE.

EMPLOYEE table:

EMP_ID	**EMP_NAME**	**EMP_PHONE**	**EMP_STATE**
14	John	7272826385, 9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389, 8589830302	Punjab

The decomposition of the EMPLOYEE table into 1NF has been shown below:

EMP_ID	**EMP_NAME**	**EMP_PHONE**	**EMP_STATE**
14	John	7272826385	UP
14	John	9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389	Punjab
12	Sam	8589830302	Punjab



Conversion of Table into First Normal Form (1NF)

ID	Name	Branch Name
1.	John	CS, Civil
2.	Ben	Electronics
3.	Steve	Mechanical
4.		
5.		

ID	Name	Branch Name
1.	John	CS
2.	John	Civil
3.	Ben	Electronics
4.	Steve	Mechanical
5.		

Second Normal Form (2NF)

- In the 2NF, relational must be in 1NF.
- In the second normal form, all non-key attributes are fully functional dependent on the primary key

Example: Let's assume, a school can store the data of teachers and the subjects they teach. In a school, a teacher can teach more than one subject.

TEACHER table

TEACHER_ID	**SUBJECT**	**TEACHER_AGE**
25	Chemistry	30
25	Biology	30
47	English	35
83	Math	38
83	Computer	38

In the given table, non-prime attribute TEACHER_AGE is dependent on TEACHER_ID which is a proper subset of a candidate key. That's why it violates the rule for 2NF.

To convert the given table into 2NF, we decompose it into two tables:

TEACHER_DETAIL table:

TEACHER_ID	**TEACHER_AGE**
25	30
47	35
83	38

TEACHER_SUBJECT table:

TEACHER_ID	**SUBJECT**
25	Chemistry
25	Biology

47	English
83	Math
83	Computer

Third Normal Form (3NF)

- A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency.

- 3NF is used to reduce the data duplication. It is also used to achieve the data integrity.

- If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.

A relation is in third normal form if it holds atleast one of the following conditions for every non-trivial function dependency X → Y.
1. X is a super key.

2. Y is a prime attribute, i.e., each element of Y is part of some candidate key

Example:
EMPLOYEE_DETAIL table:

EMP_ID	EMP_NAME	EMP_ZIP	EMP_STATE	EMP_CITY
222	Harry	201010	UP	Noida
333	Stephan	02228	US	Boston
444	Lan	60007	US	Chicago
555	Katharine	06389	UK	Norwich
666	John	462007	MP	Bhopal

Super key in the table above: 1. {EMP_ID}, {EMP_ID, EMP_NAME}, {EMP_ID, EMP_NAME, EMP_ZIP}....so on

Candidate key: {EMP_ID}
Non-prime attributes: In the given table, all attributes except EMP_ID are non-prime.
Here, EMP_STATE & EMP_CITY dependent on EMP_ZIP and EMP_ZIP dependent on EMP_ID. The non-prime attributes (EMP_STATE, EMP_CITY) transitively dependent on super key(EMP_ID). It violates the rule of third normal form.
That's why we need to move the EMP_CITY and EMP_STATE to the new <EMPLOYEE_ZIP> table, with EMP_ZIP as a Primary key.
EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_ZIP
222	Harry	201010
333	Stephan	02228
444	Lan	60007
555	Katharine	06389
666	John	462007

EMPLOYEE_ZIP table:

EMP_ZIP	EMP_STATE	EMP_CITY
201010	UP	Noida
02228	US	Boston
60007	US	Chicago
06389	UK	Norwich
462007	MP	Bhopal

Boyce Codd normal form (BCNF)

- BCNF is the advance version of 3NF. It is stricter than 3NF.
- A table is in BCNF if every functional dependency X → Y, X is the super key of the table.
- For BCNF, the table should be in 3NF, and for every FD, LHS is super key.

BCNF requires that each non-trivial dependency in a table is a dependency on a candidate key. This means that a table should not have non-trivial dependencies, where a non-primary key column depends on another non-primary key column. BCNF ensures that each table in a database is a separate entity and eliminates redundancies.
In simple words, if there exists a Functional Dependency X->Y in the table such that Y is a Prime Attribute and X is a non-prime attribute, then the table is not in BCNF

Example: Let's assume there is a company where employees work in more than one department.
EMPLOYEE table:

EMP_ID	EMP_COUNTRY	EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
264	India	Designing	D394	283
264	India	Testing	D394	300
364	UK	Stores	D283	232
364	UK	Developing	D283	549

In the above table Functional dependencies are as follows:

1. EMP_ID → EMP_COUNTRY
2. EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Candidate key: {EMP-ID, EMP-DEPT}
The table is not in BCNF because neither EMP_DEPT nor EMP_ID alone are keys.
To convert the given table into BCNF, we decompose it into three tables:
EMP_COUNTRY table:

EMP_ID	EMP_COUNTRY
264	India
264	India

EMP_DEPT table:

EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
Designing	D394	283

Testing	D394	300
Stores	D283	232
Developing	D283	549

EMP_DEPT_MAPPING table:

EMP_ID	**EMP_DEPT**
D394	283
D394	300
D283	232
D283	549

Functional dependencies:

1. EMP_ID → EMP_COUNTRY
2. EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Candidate keys:
For the first table: EMP_ID **For the second table:** EMP_DEPT **For the third table:** {EMP_ID, EMP_DEPT}
Now, this is in BCNF because left side part of both the functional dependencies is a key.

Fourth normal form (4NF)

- A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency.

- For a dependency A → B, if for a single value of A, multiple values of B exists, then the relation will be a multi-valued dependency.

4NF builds on BCNF by requiring that a table should not have multi-valued dependencies. A multi-valued dependency occurs when a non-primary key column depends on a combination of other non-primary key columns.

Example

STUDENT

STU_ID	**COURSE**	**HOBBY**
21	Computer	Dancing
21	Math	Singing
34	Chemistry	Dancing
74	Biology	Cricket
59	Physics	Hockey

The given STUDENT table is in 3NF, but the COURSE and HOBBY are two independent entity. Hence, there is no relationship between COURSE and HOBBY.

In the STUDENT relation, a student with STU_ID, **21** contains two courses, **Computer** and **Math** and two hobbies, **Dancing** and **Singing**. So there is a Multi-valued dependency on STU_ID, which leads to unnecessary repetition of data.

So to make the above table into 4NF, we can decompose it into two tables:

STUDENT_COURSE

STU_ID	COURSE
21	Computer
21	Math
34	Chemistry
74	Biology
59	Physics

STUDENT_HOBBY

STU_ID	HOBBY
21	Dancing
21	Singing
34	Dancing
74	Cricket
59	Hockey

Fifth normal form (5NF)

- A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.

- 5NF is satisfied when all the tables are broken into as many tables as possible in order to avoid redundancy.

- 5NF is also known as Project-join normal form (PJ/NF).

Example

SUBJECT	LECTURER	SEMESTER
Computer	Anshika	Semester 1
Computer	John	Semester 1
Math	John	Semester 1
Math	Akash	Semester 2
Chemistry	Praveen	Semester 1

In the above table, John takes both Computer and Math class for Semester 1 but he doesn't take Math class for Semester 2. In this case, combination of all these fields required to identify a valid data. Suppose we add a new Semester as Semester 3 but do not know about the subject and who will be taking that subject so we leave Lecturer and Subject as NULL. But all three columns together acts as a primary key, so we can't leave other two columns blank.

So to make the above table into 5NF, we can decompose it into three relations P1, P2 & P3:

P1

SEMESTER	**SUBJECT**
Semester 1	Computer
Semester 1	Math
Semester 1	Chemistry
Semester 2	Math

P2

SUBJECT	**LECTURER**
Computer	Anshika
Computer	John
Math	John
Math	Akash
Chemistry	Praveen

P3

SEMSTER	**LECTURER**
Semester 1	Anshika
Semester 1	John
Semester 1	John
Semester 2	Akash
Semester 1	Praveen

Practicing DDL commands

SQL DDL Commands:
The DDL commands are
1) Create
2) Alter
3) Drop
4) Truncate
5) Rename

1) Creation of Table:

This command is used for creating tables.

Syntax:

```
create table tablename(column-name data-type constraints….);
```

```
mysql> create table student(name varchar(30),id int primary key,address
varchar(50),marks int);

Query OK, 0 rows affected

mysql> insert into student values('rani',201,'Hyderabad',50);

Query OK, 1 row affected

mysql> insert into student values('raju',202,'Delhi',55);

Query OK, 1 row affected

mysql> insert into student values('shilpa',203,'Pune',60);

Query OK, 1 row affected

mysql> insert into student values('ram',204,'Chenni',70),('ravi',205,'Bombay',40);

Query OK, 2 rows affected
```

```
mysql> select * from student;

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

| name   | id  | address   | marks |

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

| rani   | 201 | Hyderabad |    50 |

| raju   | 202 | Delhi     |    55 |

| shilpa | 203 | Pune      |    60 |

| ram    | 204 | Chenni    |    70 |

| ravi   | 205 | Bombay    |    40 |

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

5 rows
```

2) Altering the Table:

It is used for modifying the table structure.

```
mysql> alter table student add phonenumber int;

Query OK, 0 rows affected

mysql> select*from student;

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

| name   | id  | address   | marks | phonenumber |

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

| rani   | 201 | Hyderabad |    50 |        NULL |
| raju   | 202 | Delhi     |    55 |        NULL |
| shilpa | 203 | Pune      |    60 |        NULL |
| ram    | 204 | Chenni    |    70 |        NULL |
| ravi   | 205 | Bombay    |    40 |        NULL |
+--------+-----+-----------+-------+------------+
5 rows
```

4) Truncate the Table:

It is used for deleting the data in the table but the table structure exists.

```
mysql> truncate table student;
Query OK, 0 rows affected
mysql> select*from student;
Empty set
```

5) Renaming of the Table:

It is used for changing the existing table names.

```
mysql>rename student to student1;
```

6) Dropping of the Table;

It is used for deleting the table structure and data permanently.

```
mysql> drop table student;
Query OK, 0 rows affected
mysql> select*from student;
ERROR 1146 (42S02): Table 'divya.student' doesn't exist
```

Experiment 3: Practicing DML commands

DML commands are used for managing data with in schema objects.
Few DML commands are
1) select
2) insert
3) update
4) delete

1) select:

This command is used to retrieve data from the table.
Syntax:

```
Select * from table name;
```

```
mysql> select *from student;

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

| name | id  | address   | marks |

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

| rani | 201 | Hyderabad |    50 |

| raju | 202 | Delhi     |    55 |

| ram  | 204 | Chenni    |    70 |

| ravi | 205 | Bombay    |    40 |

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

2) Insert:

This command is used to insert the data into database.

Creation of bus table:

```
mysql>  create table bus(busnumber varchar(20),source varchar(20), destination varchar(20));

Query OK, 0 rows affected

mysql> insert into bus values (1234,'hyd', 'tirupathi');

Query OK, 1 row affected
```

3) Update:

This command is used to update existing data with in a table.

Commmand:

```
update bus destination ‘banglore’ where bus no=23;
```

4) Delete:

Delete command is used to delete the records (or) rows (or) complete table from the database

Command:

```
delete from bus where busnumber='1234';

Query OK, 1 row affected
```

5. Querying (using ANY, ALL, UNION, INTERSECT, JOIN Constraints etc.)

UNION

Union is used to combine the results of two queries into a single result set of all matching rows. Both the queries must result in the same number of columns and compatible data types in order to unite. All duplicate records are removed automatically unless UNION ALL is used.

Syntax

```
{ <query_specification> | ( <query_expression> ) }
{ UNION | UNION ALL}
{ <query_specification> | ( <query_expression> ) }
```

```
mysql> create table authors(name varchar(30));
Query OK,
mysql> insert into authors values('Rohith'),('kavya'),('Rahul');
Query OK, 3 rows affected
mysql> select * from authors;
+--------+
| name   |
+--------+
| Rohith |
| kavya  |
| Rahul  |
+--------+
3 rows in set
mysql> create table speakers(name varchar(30));
Query OK,
mysql> insert into speakers values('Rani'),('Raju'),('Shilpa'),('sony');
Query OK,
mysql> select * from speakers;
+--------+
| name   |
+--------+
| Rani   |
| Raju   |
| Shilpa |
| sony   |
+--------+

mysql> select name from Speakers union select name from Authors order by
name;
```

```
+--------+
| name   |
+--------+
| kavya  |
| Rahul  |
| Raju   |
| Rani   |
| Rohith |
| Shilpa |
| sony   |
+--------+
```

INTERSECT

It is used to take the result of two queries and returns only those rows which are common in both result sets. It removes duplicate records from the final result set.

Syntax

```
{ <query_specification> | ( <query_expression> ) }
{  INTERSECT }
{ <query_specification> | ( <query_expression> ) }
```

You want the list of people who are Speakers and they are also Authors. Hence, how will you prepare such a list?

```
mysql> select name from Speakers intersect select name from Authors order by name;
Empty set
```

EXCEPT / MINUS

It is used to take the distinct records of two one query and returns only those rows which do not appear in the second result set.

Syntax

```
{ <query_specification> | ( <query_expression> ) }
{ EXCEPT | INTERSECT }
{ <query_specification> | ( <query_expression> ) }
```

You want the list of people who are only Speakers and they are not Authors. Hence, how will you prepare such a list?

```
mysql> select name from Speakers except select name from Authors order by name;
```

```
+--------+
| name   |
+--------+
| Raju   |
| Rani   |
| Shilpa |
| sony   |
+--------+
```

You want the list of people who are only Authors and they are not Speakers. Hence, how will you prepare such a list?

```
mysql> select name from Authors except select name from Speakers order by name;
+--------+
| name   |
+--------+
| kavya  |
| Rahul  |
| Rohith |
+--------+
3 rows in set
```

The SQL ANY Operator

The ANY operator:

- returns a boolean value as a result
- returns TRUE if ANY of the subquery values meet the condition

ANY means that the condition will be true if the operation is true for any of the values in the range.

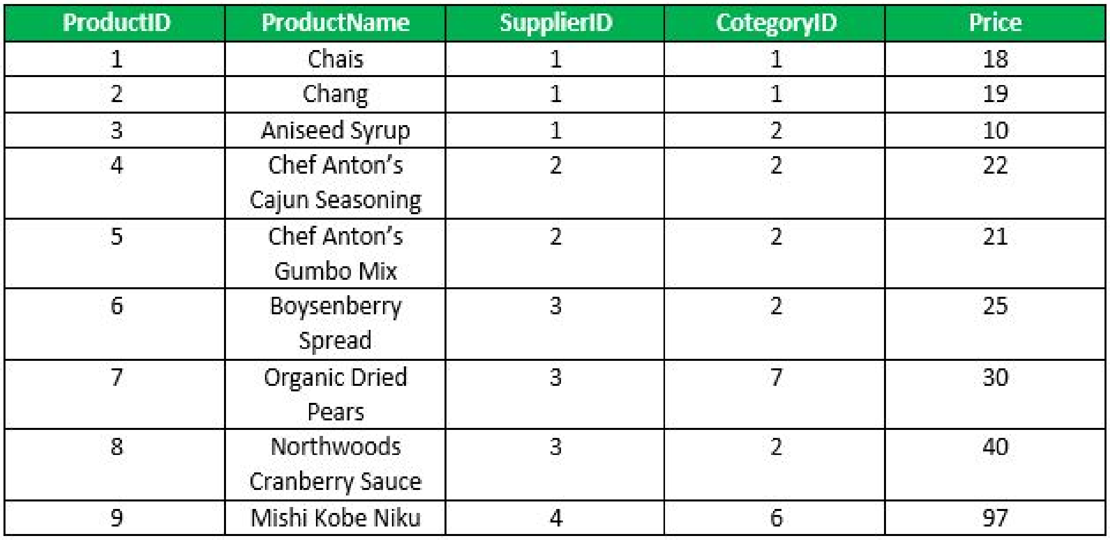
ANY Syntax

```
SELECT column_name(s)
 FROM table_name
WHERE column_name operator ANY
 (SELECT column_name
 FROM table_name
 WHERE condition);
```

Note: The *operator* must be a standard comparison operator (=, <>, !=, >, >=, <, or <=).

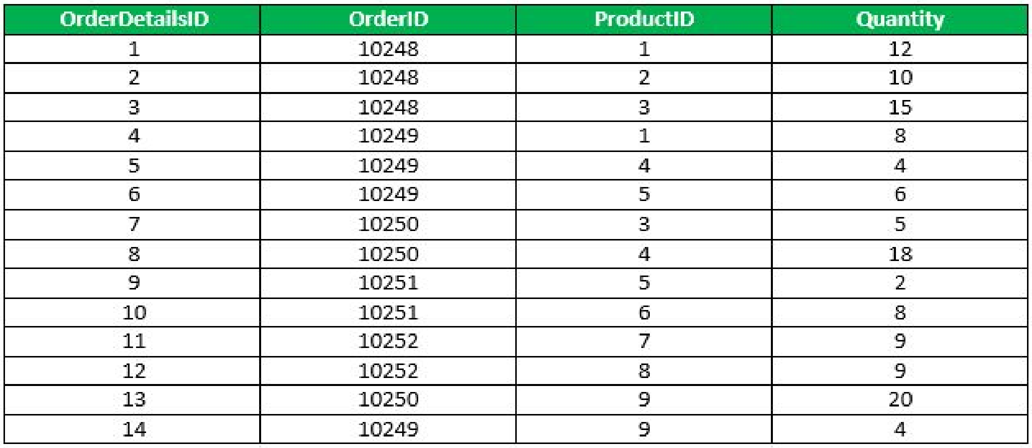
Create Table

Consider the following Products Table and OrderDetails Table,**Products Table**



ProductID	ProductName	SupplierID	CotegoryID	Price
1	Chais	1	1	18
2	Chang	1	1	19
3	Aniseed Syrup	1	2	10
4	Chef Anton's Cajun Seasoning	2	2	22
5	Chef Anton's Gumbo Mix	2	2	21
6	Boysenberry Spread	3	2	25
7	Organic Dried Pears	3	7	30
8	Northwoods Cranberry Sauce	3	2	40
9	Mishi Kobe Niku	4	6	97

OrderDetails Table



OrderDetailsID	OrderID	ProductID	Quantity
1	10248	1	12
2	10248	2	10
3	10248	3	15
4	10249	1	8
5	10249	4	4
6	10249	5	6
7	10250	3	5
8	10250	4	18
9	10251	5	2
10	10251	6	8
11	10252	7	9
12	10252	8	9
13	10250	9	20
14	10249	9	4

SQL ANY Examples

1) The following SQL statement lists the ProductName if it finds ANY records in the OrderDetails table has Quantity equal to 10 (this will return TRUE because the Quantity column has some values of 10):

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY
(SELECT ProductID
FROM OrderDetails
WHERE QuantityID = 10);
```

```
+-------------+
| ProductName |
+-------------+
| chang       |
+-------------+
1 row in set
```

2) The following SQL statement lists the ProductName if it finds ANY records in the OrderDetails table has Quantity larger than 99 (this will return TRUE because the Quantity column has some values larger than 99):

```
mysql> SELECT ProductName FROM Products WHERE ProductID = ANY (SELECT
ProductID FROM OrderDetails WHERE QuantityID > 10);
```

```
+-----------------------------+
| ProductName                 |
+-----------------------------+
| chais                       |
| Aniseed syrup               |
| chef antnons cajun seasoning |
| mishikobeniku               |
+-----------------------------+
4 rows in set
```

The SQL ALL Operator

The ALL operator:

- returns a boolean value as a result
- returns TRUE if ALL of the subquery values meet the condition
- is used with SELECT, WHERE and HAVING statements

ALL means that the condition will be true only if the operation is true for all values in the range.

ALL Syntax With SELECT

```
SELECT ALL column_name(s)
FROM table_name
WHERE condition;
```

ALL Syntax With WHERE or HAVING

```
SELECT column_name(s)
 FROM table_name
WHERE column_name operator ALL (SELECT column_name FROM table_name WHERE
condition);
```

Note: The *operator* must be a standard comparison operator (=, <>, !=, >, >=, <, or <=).

1) **The following SQL statement lists ALL the product names:**

```
mysql> SELECT ALL ProductName FROM Products WHERE TRUE;
+-----------------------------+
| ProductName                 |
+------------------------------+
| chais                       |
| chang                       |
| Aniseed syrup               |
| chef antnons cajun seasoning |
| chefantonsgumbo mix         |
| boysenberry spread          |
| orgnicdriedpears            |
| northwoodscranberrysauce    |
| mishikobeniku               |
+-----------------------------+
9 rows in set
```

2) **The following SQL statement lists the ProductName if ALL the records in the OrderDetails table has Quantity equal to 10. This will of course return FALSE because the Quantity column has many different values (not only the value of 10):**

```
mysql> SELECT ProductName FROM Products WHERE ProductID = ALL (SELECT
ProductID FROM OrderDetails WHERE QuantityID = 10);
+-------------+
| ProductName |
+-------------+
| chang       |
+-------------+
1 row in set
```

8) Queries using Aggregate functions, GROUP BY, HAVING and Creation and dropping of Views

GROUP BY Syntax

```
SELECT column1, column2, ..., columnN, aggregate_function(column_Z)
FROM table_name
WHERE condition
GROUP BY column1, column2, ..., columnN;
```

```
mysql>  CREATE TABLE Orders ( id INTEGER NOT NULL AUTO_INCREMENT PRIMARY
KEY, cust_id INTEGER, amount INTEGER NOT NULL );
Query OK
```

```
mysql>  INSERT INTO Orders(cust_id,amount) VALUES (1,105); INSERT INTO
Orders(cust_id,amount) VALUES (1,78); INSERT INTO Orders(cust_id,amount) VALUES (3,55);
INSERT INTO Orders(cust_id,amount) VALUES (3,42); INSERT INTO Orders(cust_id,amount)
VALUES (2,215);
Query OK, 1 row affected (0.23 sec)

Query OK, 1 row affected
Query OK, 1 row affected
Query OK, 1 row affected
Query OK, 1 row affected
```

```
mysql> SELECT * FROM Orders;
+----+---------+--------+
| id | cust_id | amount |
+----+---------+--------+
|  1 |       1 |    105 |
|  2 |       1 |     78 |
|  3 |       3 |     55 |
|  4 |       3 |     42 |
|  5 |       2 |    215 |
+----+---------+--------+
5 rows in set (0.05 sec)
mysql> SELECT cust_id, SUM(amount) as total_amount FROM Orders GROUP BY cust_id;
+---------+--------------+
| cust_id | total_amount |
+---------+--------------+
|       1 |          183 |
|       3 |           97 |
|       2 |          215 |
+---------+--------------+
3 rows in set
```

- You can also use expressions in the GROUP BY clause to group data based on calculated values. Consider the following query on the same Orders table to understand the usage of GROUP BY with an expression example

```
mysql> SELECT CASE
 WHEN amount < 50 THEN 'Low'
WHEN amount >= 50 AND amount < 150 THEN 'Medium'
WHEN amount >= 150 THEN 'High'
 END as amount_range,
COUNT(*) as count_orders
FROM Orders
GROUP BY amount_range;

+--------------+--------------+
| amount_range | count_orders |
+--------------+--------------+
| Medium       |            3 |
| Low          |            1 |
| High         |            1 |
+--------------+--------------+
3 rows in set
```

GROUP BY With HAVING Clause

```
mysql> SELECT cust_id, SUM(amount) as total_amount
FROM Orders
GROUP BY cust_id
HAVING total_amount <= 200;

+---------+--------------+
| cust_id | total_amount |
+---------+--------------+
|       1 |          183 |
|       3 |           97 |
+---------+--------------+
2 rows in set
```

Example

Let's create a view in MySQL Command line client using an existing table inside a database. Suppose, we have a table EMPLOYEE which displays the details of employees working in an organization.

Emp_id	**first_name**	**last_name**	**Emp_age**	**Emp_salary**
101	Harry	Wills	29	20000
102	Nicholas	Byer	27	30000
103	Marie	Curie	24	55000
104	Karl	Anderson	38	70000

Query:

```
CREATE VIEW EMP_DETAILS AS
SELECT Emp_id,Emp_salary
FROM EMPLOYEE
WHERE Emp_salary > 20000;
SELECT * FROM EMP_DETAILS;
```

Output:

```
Emp_id              Emp_salary
102                 30000
103                 55000
104                 70000
```

```
mysql> create database triggers;

mysql> use triggers;

mysql> show tables;

Empty set
```

#Before Insert

```
mysql> create table customers(cust_id int,age int,name varchar(30));

Query OK, 0 rows affected

mysql> delimiter //

mysql> create trigger age_verify

    -> before insert on customers

    -> for each row

    -> if new.age < 0 then set new.age = 0;

    -> end if; //

Query OK, 0 rows affected

mysql> insert into customers values(101,27,"james"), (102,-40,"Rani"), (103,32,"Ben"), (104, -39,"Raju");

Query OK, 4 rows affected


mysql> select * from customers;

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

| cust_id | age  | name  |

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

|     101 |   27 | james |

|     102 |    0 | Rani  |

|     103 |   32 | Ben   |

|     104 |    0 | Raju  |
+---------+------+-------+

4 rows in set
```

# After Insert

```
mysql> create table customers1(id int auto_increment primary key,name varchar(40) not null,email varchar(30), birthdate date);

Query OK, 0 rows affected

mysql> create table message(id int auto_increment,messageId int, message varchar(300) not null, primary key(id,messageId));

Query OK, 0 rows affected

mysql> delimiter //

mysql> create trigger check_null_dob

    -> after insert on customers1

    -> for each row

    -> begin

    -> if new.birthdate is null then

    -> insert into message(messageId,message)

    -> values(new.id,concat("Hi",new.name,'please update your date of birth'));

    -> end if;

    -> end //

Query OK, 0 rows affected (0.26 sec)


mysql> delimiter ;

mysql> insert into customers1(name, email, birthdate)
```

```
    -> values("Nani","nani@abc.com",null),("Rani","Rani@abc.com","1999-11-16"),("Anu","Anu@abc.com","1997-08-20"),("Raju","Raju@abc.com",null);

Query OK, 4 rows affected

mysql> select * from message;

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

| id | messageId | message                              |

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

|  1 |         1 | HiNaniplease update your date of birth |

|  2 |         4 | HiRajuplease update your date of birth |

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

2 rows in set
```

# Before Update

```
mysql> create table employees(emp_id int primary key,emp_name varchar(30),age int,salary float);

Query OK, 0 rows affected

mysql> insert into employees values(101,"James",35,70000),(102,"Sony",30,55000),(103,"Marry",28,62000),(104,"Raju",37,37000),(105,"Rani",32,57000),(106,"Anu",35,8000),(107,"Jack",40,100000);

Query OK, 7 rows affected

mysql> delimiter //

mysql> create trigger update_trigger
```

```
    -> before update
    -> on employees
    -> for each row
    -> begin
    -> if new.salary=10000 then
    -> set new.salary = 85000;
    -> elseif new.salary <10000 then
    -> set new.salary = 72000;
    -> end if;
    -> end //
Query OK, 0 rows affected
mysql> delimiter ;
mysql> update employees
    -> set salary = 8000;
Query OK, 7 rows affected
mysql> select * from employees;
+--------+----------+------+--------+
| emp_id | emp_name | age  | salary |
+--------+----------+------+--------+
|    101 | James    |   35 |  72000 |
|    102 | Sony     |   30 |  72000 |
|    103 | Marry    |   28 |  72000 |
|    104 | Raju     |   37 |  72000 |
|    105 | Rani     |   32 |  72000 |
|    106 | Anu      |   35 |  72000 |
|    107 | Jack     |   40 |  72000 |
+--------+----------+------+--------+
7 rows in set
```

# Before Delete

```
create table salarydel(id int primary key auto_increment,eid int, validfrom date not null,amount float not null, deletedat timestamp default now());
Query OK, 0 rows affected
mysql> delimiter $$
mysql> create trigger salary_delete
    -> before delete
    -> on salary
    -> for each row
    -> begin
    -> insert into salarydel(eid,validfrom,amount)
    -> value(old.eid, old.validfrom, old.amount);
    -> end$$
mysql> delimiter;
delete from salary
```

```
where eid = 103;

select *from salarydel;
```

Procedures

A procedure (often called a stored procedure) is a collection of pre-compiled SQL statements stored inside the database.
A procedure always contains a name, parameter lists, and SQL statements.

Syntax

```
DELIMITER $$
CREATE PROCEDURE procedure_name [[IN | OUT | INOUT] parameter_name datatype [, parameter datatype]) ]
BEGIN
Declaration_section
Executable_section
END $$
DELIMITER ;
```

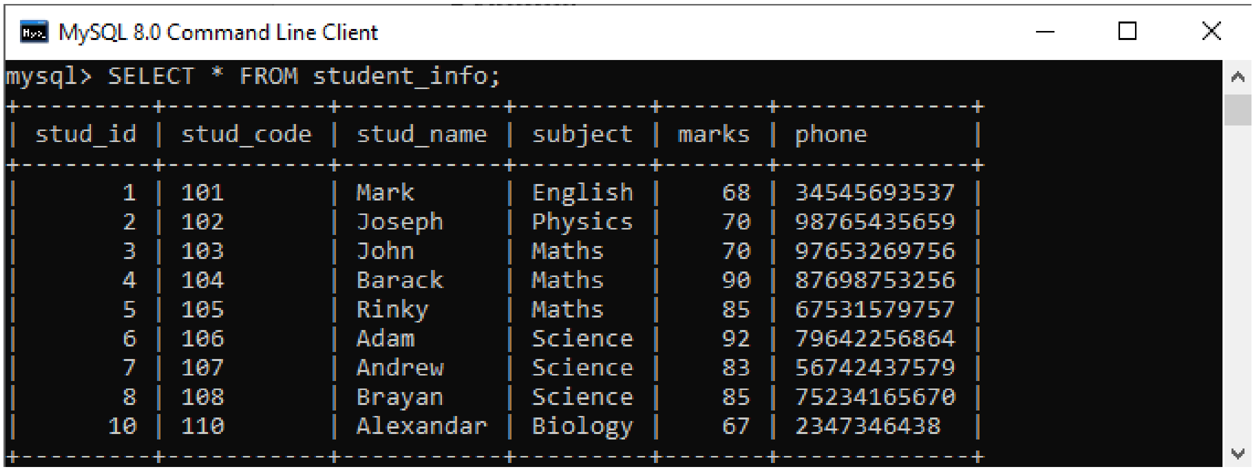
Parameter Name	**Descriptions**
procedure_name	It represents the name of the stored procedure.
parameter	It represents the number of parameters. It can be one or more than one.
Declaration_section	It represents the declarations of all variables.
Executable_section	It represents the code for the function execution

MySQL procedure parameter has one of three modes:

IN parameter
It is the default mode. It takes a parameter as input, such as an attribute. When we define it, the calling program has to pass an argument to the stored procedure. This parameter's value is always protected.

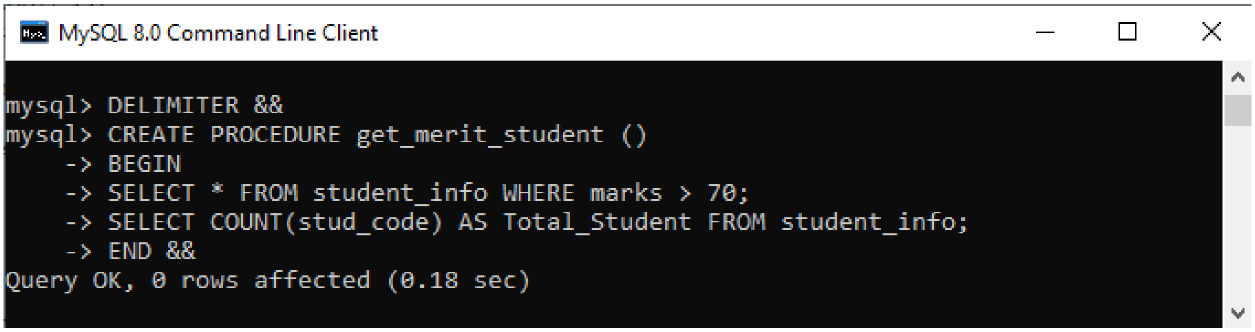
OUT parameters
It is used to pass a parameter as output. Its value can be changed inside the stored procedure, and the changed (new) value is passed back to the calling program. It is noted that a procedure cannot access the OUT parameter's initial value when it starts.

INOUT parameters
It is a combination of IN and OUT parameters. It means the calling program can pass the argument, and the procedure can modify the INOUT parameter, and then passes the new value back to the calling program.

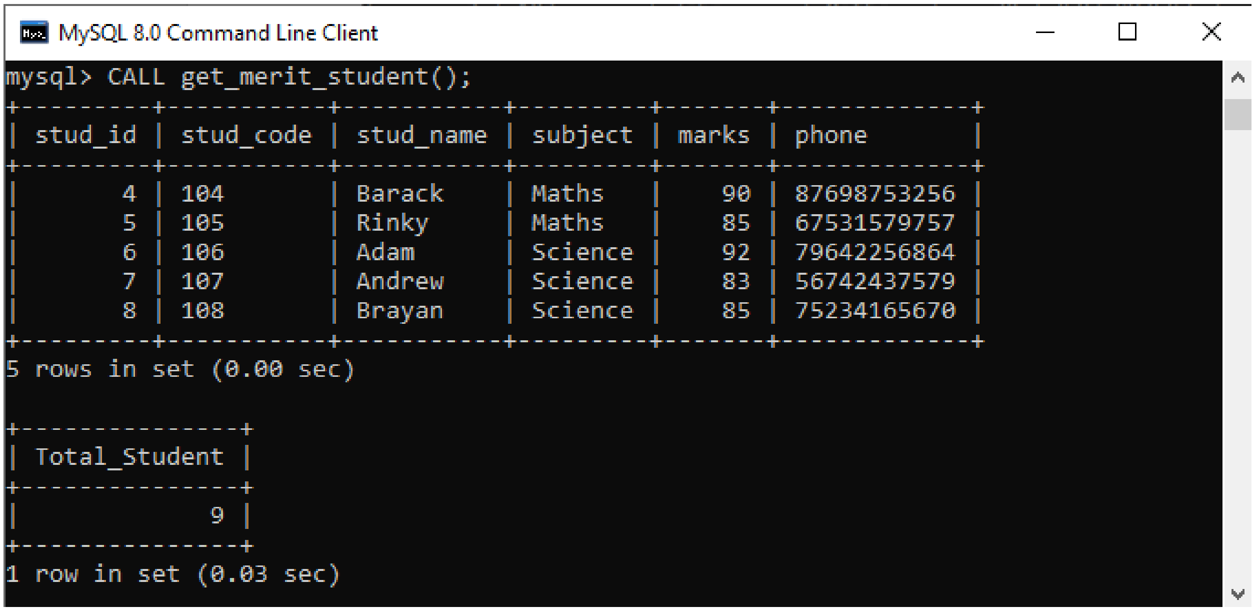


```
mysql> SELECT * FROM student_info;
+---------+-----------+-----------+---------+-------+-------------+
| stud_id | stud_code | stud_name | subject | marks | phone       |
+---------+-----------+-----------+---------+-------+-------------+
|       1 | 101       | Mark      | English |    68 | 34545693537 |
|       2 | 102       | Joseph    | Physics |    70 | 98765435659 |
|       3 | 103       | John      | Maths   |    70 | 97653269756 |
|       4 | 104       | Barack    | Maths   |    90 | 87698753256 |
|       5 | 105       | Rinky     | Maths   |    85 | 67531579757 |
|       6 | 106       | Adam      | Science |    92 | 79642256864 |
|       7 | 107       | Andrew    | Science |    83 | 56742437579 |
|       8 | 108       | Brayan    | Science |    85 | 75234165670 |
|      10 | 110       | Alexandar | Biology |    67 | 2347346438  |
+---------+-----------+-----------+---------+-------+-------------+
```

Procedure without Parameter



```
mysql> DELIMITER &&
mysql> CREATE PROCEDURE get_merit_student ()
    -> BEGIN
    -> SELECT * FROM student_info WHERE marks > 70;
    -> SELECT COUNT(stud_code) AS Total_Student FROM student_info;
    -> END &&
Query OK, 0 rows affected (0.18 sec)
```

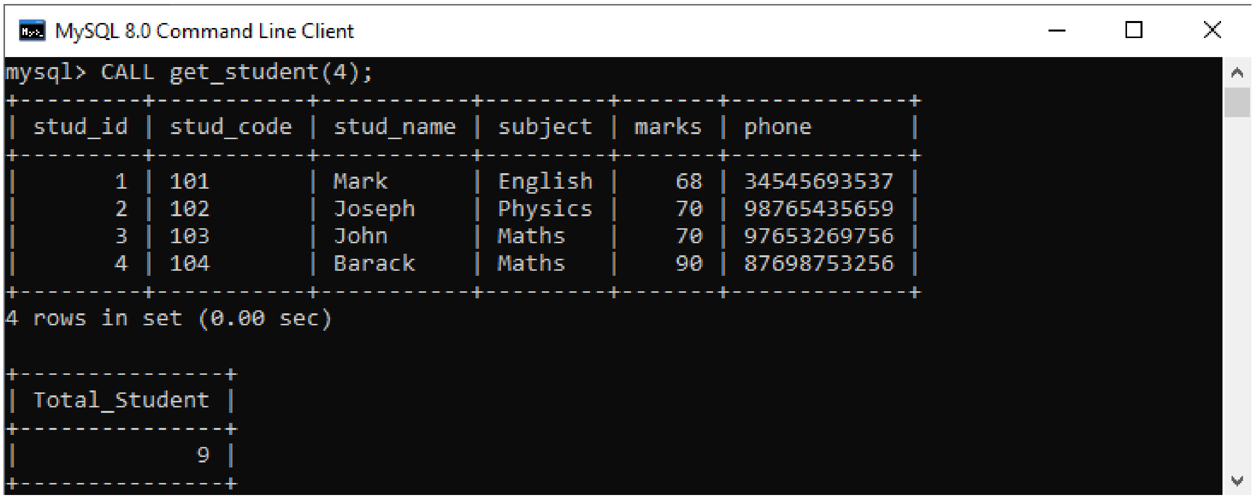


```
mysql> CALL get_merit_student();
+---------+-----------+-----------+---------+-------+-------------+
| stud_id | stud_code | stud_name | subject | marks | phone       |
+---------+-----------+-----------+---------+-------+-------------+
|       4 | 104       | Barack    | Maths   |    90 | 87698753256 |
|       5 | 105       | Rinky     | Maths   |    85 | 67531579757 |
|       6 | 106       | Adam      | Science |    92 | 79642256864 |
|       7 | 107       | Andrew    | Science |    83 | 56742437579 |
|       8 | 108       | Brayan    | Science |    85 | 75234165670 |
+---------+-----------+-----------+---------+-------+-------------+
5 rows in set (0.00 sec)

+---------------+
| Total_Student |
+---------------+
|             9 |
+---------------+
1 row in set (0.03 sec)
```

Procedures with IN Parameter

```
DELIMITER $$
CREATE PROCEDURE get_student (IN var1 INT)
BEGIN
SELECT * FROM student_info LIMIT var1;
SELECT COUNT(stud_code) AS Total_Student FROM student_info;
END $$
DELIMITER ;
```

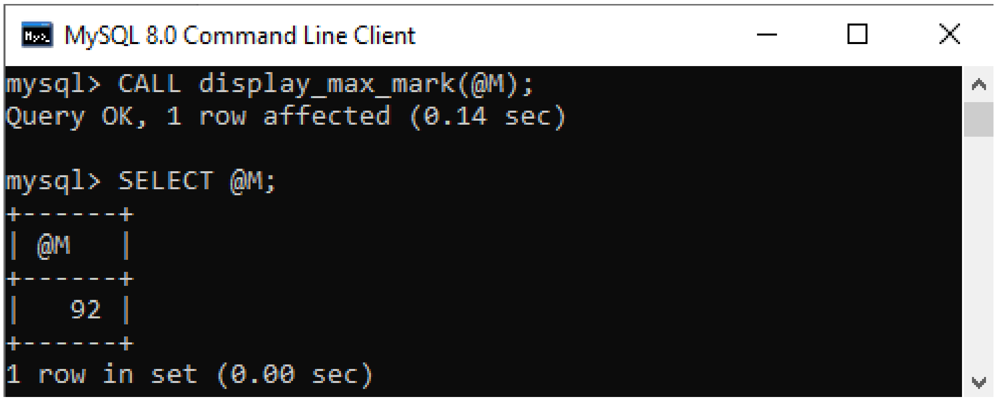


```
mysql> CALL get_student(4);
+---------+-----------+-----------+---------+-------+-------------+
| stud_id | stud_code | stud_name | subject | marks | phone       |
+---------+-----------+-----------+---------+-------+-------------+
|       1 | 101       | Mark      | English |    68 | 34545693537 |
|       2 | 102       | Joseph    | Physics |    70 | 98765435659 |
|       3 | 103       | John      | Maths   |    70 | 97653269756 |
|       4 | 104       | Barack    | Maths   |    90 | 87698753256 |
+---------+-----------+-----------+---------+-------+-------------+
4 rows in set (0.00 sec)

+---------------+
| Total_Student |
+---------------+
|             9 |
+---------------+
```

Procedures with OUT Parameter

```
DELIMITER $$
CREATE PROCEDURE display_max_mark (OUT highestmark INT)
BEGIN
SELECT MAX(marks) INTO highestmark FROM student_info;
END $$
DELIMITER ;
```

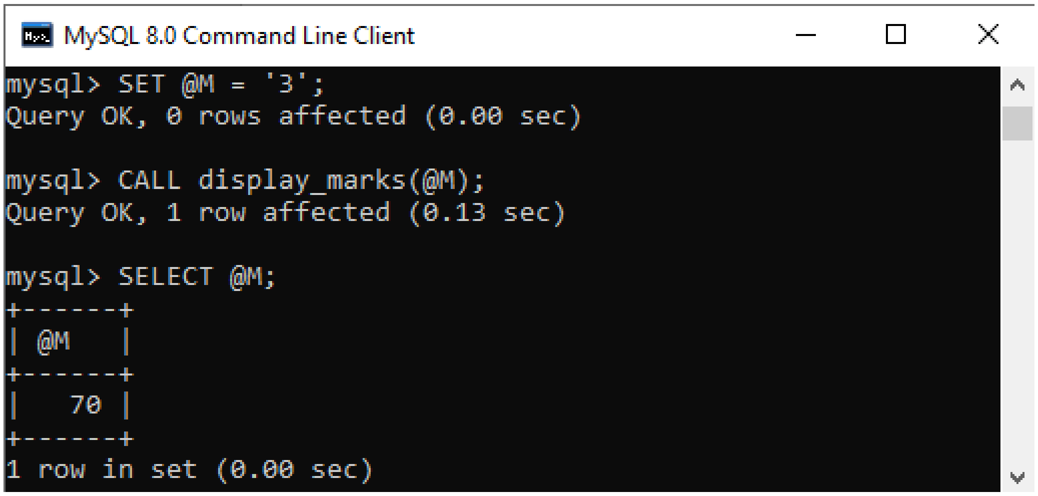


```
mysql> CALL display_max_mark(@M);
Query OK, 1 row affected (0.14 sec)

mysql> SELECT @M;
+------+
| @M   |
+------+
|   92 |
+------+
1 row in set (0.00 sec)
```

Procedures with INOUT Parameter

```
DELIMITER $$
CREATE PROCEDURE display_marks (INOUT var1 INT)
BEGIN
SELECT marks INTO var1 FROM student_info WHERE stud_id = var1;
END $$
DELIMITER ;
```



```
mysql> SET @M = '3';
Query OK, 0 rows affected (0.00 sec)

mysql> CALL display_marks(@M);
Query OK, 1 row affected (0.13 sec)

mysql> SELECT @M;
+------+
| @M   |
+------+
|   70 |
+------+
1 row in set (0.00 sec)
```

MySQL Cursor

In MySQL, Cursor can also be created. Following are the steps for creating a cursor.

1. Declare Cursor

A cursor is a select statement, defined in the declaration section in MySQL.

Syntax

```
DECLARE cursor_name CURSOR FOR
Select statement;
```

Parameter:

cursor_name: name of the cursor

select_statement: select query associated with the cursor

2. Open Cursor

After declaring the cursor the next step is to open the cursor using open statement.

Syntax

```
Open cursor_name;
```

Parameter:

cursor_name: name of the cursor which is already declared.

3. Fetch Cursor

After declaring and opening the cursor, the next step is to fetch the cursor. It is used to fetch the row or the column.

Syntax

```
FETCH [ NEXT [ FROM ] ] cursor_name INTO variable_list;
```

Parameter:

cursor_name: name of the cursor

variable_list: variables, comma separated, etc. is stored in a cursor for the result set

4. Close Cursor

The final step is to close the cursor.

Syntax

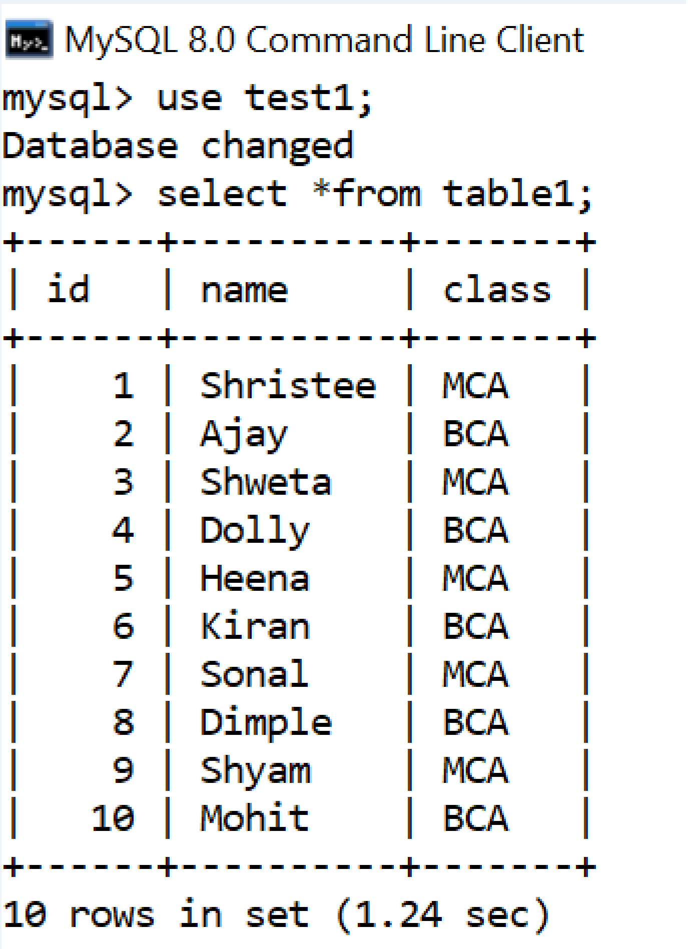
1. Close cursor_name;

Parameter:

Cursor_name: name of the cursor

Example for the cursor:

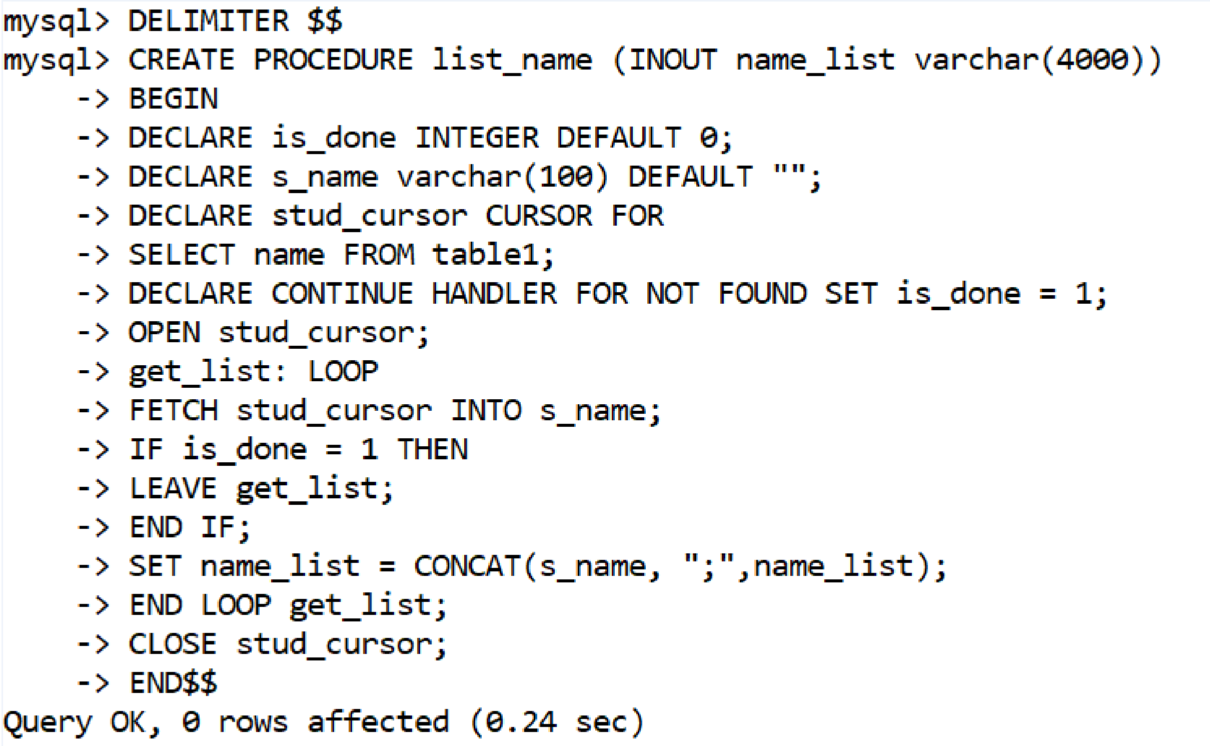
Step 1: Open the database and table.



```
mysql> use test1;
Database changed
mysql> select *from table1;
+------+----------+-------+
| id   | name     | class |
+------+----------+-------+
|    1 | Shristee | MCA   |
|    2 | Ajay     | BCA   |
|    3 | Shweta   | MCA   |
|    4 | Dolly    | BCA   |
|    5 | Heena    | MCA   |
|    6 | Kiran    | BCA   |
|    7 | Sonal    | MCA   |
|    8 | Dimple   | BCA   |
|    9 | Shyam    | MCA   |
|   10 | Mohit    | BCA   |
+------+----------+-------+
10 rows in set (1.24 sec)
```

Step 2: Now create the cursor.

Query:

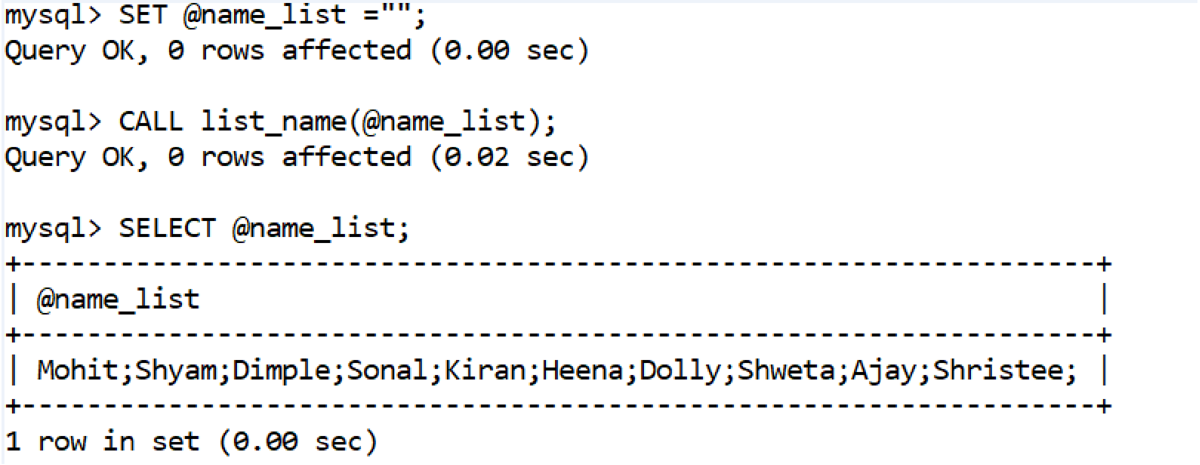


```
mysql> DELIMITER $$
mysql> CREATE PROCEDURE list_name (INOUT name_list varchar(4000))
    -> BEGIN
    -> DECLARE is_done INTEGER DEFAULT 0;
    -> DECLARE s_name varchar(100) DEFAULT "";
    -> DECLARE stud_cursor CURSOR FOR
    -> SELECT name FROM table1;
    -> DECLARE CONTINUE HANDLER FOR NOT FOUND SET is_done = 1;
    -> OPEN stud_cursor;
    -> get_list: LOOP
    -> FETCH stud_cursor INTO s_name;
    -> IF is_done = 1 THEN
    -> LEAVE get_list;
    -> END IF;
    -> SET name_list = CONCAT(s_name, ";",name_list);
    -> END LOOP get_list;
    -> CLOSE stud_cursor;
    -> END$$
Query OK, 0 rows affected (0.24 sec)
```

Step 3: Now call the cursor.

Query:

1. SET @name_list ="";
2. CALL list_name(@name_list);
3. SELECT @name_list;



```
mysql> SET @name_list ="";
Query OK, 0 rows affected (0.00 sec)

mysql> CALL list_name(@name_list);
Query OK, 0 rows affected (0.02 sec)

mysql> SELECT @name_list;
+--------------------------------------------------------------------+
| @name_list                                                         |
+--------------------------------------------------------------------+
| Mohit;Shyam;Dimple;Sonal;Kiran;Heena;Dolly;Shweta;Ajay;Shristee;   |
+--------------------------------------------------------------------+
1 row in set (0.00 sec)
```