

# Database

---

DOCUMENT PREPARED BY BRIJESH KAMANI

Brijesh Kamani

FULL STACK DEVELOPER TRAINEE | RKIT

## Contents

<b>Fundamentals .....</b>	<b>3</b>
<b>What is MySQL? .....</b>	<b>3</b>
<b>Design .....</b>	<b>3</b>
<b>Normalization .....</b>	<b>3</b>
<b>First Normal Form (1NF) .....</b>	<b>3</b>
<b>Second Normal Form (2NF) .....</b>	<b>5</b>
<b>Third Normal Form (3NF) .....</b>	<b>6</b>
<b>MYSQL Storage Engine .....</b>	<b>8</b>
<b>InnoDB .....</b>	<b>8</b>
<b>MyISAM .....</b>	<b>8</b>
<b>SQL Basic: .....</b>	<b>9</b>
<b>Create, Select, Insert, Update and Delete Operations .....</b>	<b>9</b>
<b>Create Database (Schema) .....</b>	<b>9</b>
<b>Create Table .....</b>	<b>9</b>
<b>Insert Command .....</b>	<b>11</b>
<b>Select Command .....</b>	<b>11</b>
<b>Update Query .....</b>	<b>16</b>
<b>Delete Query .....</b>	<b>17</b>
<b>Aggregates Functions .....</b>	<b>18</b>
<b>General .....</b>	<b>22</b>
<b>Null Value &amp; Keyword in MySQL .....</b>	<b>22</b>
<b>Auto Increment .....</b>	<b>22</b>
<b>Alter, Drop &amp; rename .....</b>	<b>26</b>
<b>ALTER Command .....</b>	<b>26</b>
<b>RENAME Command .....</b>	<b>30</b>
<b>DROP Command .....</b>	<b>31</b>
<b>LIMIT keyword .....</b>	<b>32</b>
<b>Sub-Queries .....</b>	<b>33</b>
<b>1) MySQL Subquery with IN Operator .....</b>	<b>33</b>
<b>2) MySQL Subquery with NOT-IN Operator .....</b>	<b>34</b>
<b>3) MySQL Subquery in the FROM Clause .....</b>	<b>34</b>
<b>4) MySQL Correlated Subqueries .....</b>	<b>35</b>
<b>5) MySQL Subqueries with EXISTS .....</b>	<b>35</b>
<b>6) MySQL Subqueries with NOT EXISTS .....</b>	<b>36</b>
<b>Joins .....</b>	<b>36</b>

MySQL Inner JOIN (Simple Join) .....	36
MySQL Left Outer Join.....	37
MySQL Right Outer Join .....	38
Unions.....	38
Views.....	40
Index .....	42
CASE-WHEN-THEN .....	43
Stored-Procedures.....	44
Functions .....	46
Advance .....	49
Partitions .....	49
RANGE Partitioning .....	49
LIST Partitioning .....	51
HASH Partitioning.....	51
KEY Partitioning.....	51
Range Column Partitioning .....	52
List Columns Partitioning .....	52
SUBPARTITIONING .....	52
Backup .....	53

# Fundamentals

## What is MySQL?

- MySQL, the most popular Open-Source SQL database management system, is developed, distributed, and supported by Oracle Corporation.
- The SQL part of “MySQL” stands for “Structured Query Language”. SQL is the most common standardized language used to access databases.

### These Are Main Features of MySQL:

1. *MySQL databases are relational.*
2. *MySQL software is Open Source. The MySQL Database Server is very fast, reliable, scalable, and easy to use.*
3. *MySQL Server works in client/server or embedded systems.*
4. *A large amount of contributed MySQL software is available.*

## Design

### Normalization

- *Normalization is a database design technique that reduces data redundancy and eliminates undesirable characteristics like Insertion, Update and Deletion Anomalies.*
- *Normalization rules divide larger tables into smaller tables and link them using relationships.*
- *Normalisation in SQL aims to eliminate redundant (repetitive) data and ensure data is stored logically.*

### Database Normal Forms

Here is a list of Normal Forms in SQL:

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

### 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 a single-valued attribute.
- The first normal form disallows the multi-valued attribute, composite attribute, and their combinations.

**Example:** Relation EMPLOYEE is not in 1NF because of the 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

## 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, the 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 does and not contain any transitive partial dependency.
- 3NF is used to reduce data duplication. It is also used to achieve integrity.
- If there is no transitive dependency for non-prime attributes, then the real relationship is in third normal form.

#### 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) are transitively dependent on the super key (EMP\_ID). It violates the rules of the 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

## MYSQL Storage Engine

### InnoDB

- It is a transaction storage type that automatically rollbacks the writes if they are not completed.
- It uses row-level locking of the table.
- It supports the Foreign Key option.
- It supports ACID properties.
- It supports transactional properties, i.e. rollbacks and commits.

### MyISAM

- It is a non-transactional storage type, and any write option needs to be rolled back manually (if needed).
- It uses the default method of table locking and allows a single session to modify the table.
- It doesn't support the Foreign key option
- It doesn't support ACID properties
- It doesn't support transactional properties and is faster to read.

## SQL Basic:

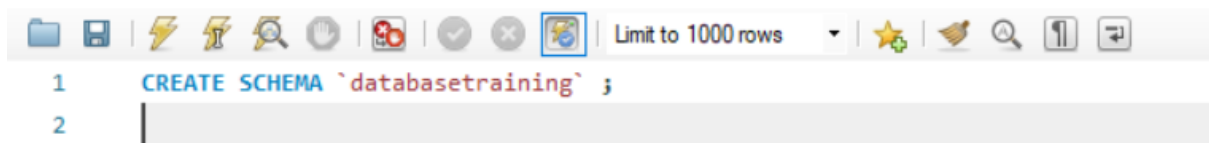
### Create, Select, Insert, Update and Delete Operations

#### Create Database (Schema)

##### Syntax:-

Create schema `database\_name`;

##### Example:-



##### Output:-

#	Time	Action	Message
✓ 1	14:08:19	CREATE SCHEMA `databasetraining`	1 row(s) affected

#### Create Table

##### Syntax:

```
CREATE TABLE [IF NOT EXISTS] table_name (  
    column_1_definition,  
    column_2_definition,  
    ...,  
    table_constraints  
) [ENGINE=storage_engine;]
```

**Example:**

```

1 CREATE TABLE `databasetraining`.`STD01` (
2   `D01F01` INT NOT NULL AUTO_INCREMENT COMMENT 'StudentID',
3   `D01F02` VARCHAR(100) NOT NULL COMMENT 'StudentName',
4   `D01F03` INT NOT NULL COMMENT 'EnrollemntNo',
5   `D01F04` TINYINT(1) NOT NULL COMMENT 'Semester',
6   `T01F01` INT NOT NULL COMMENT 'DepartmentID',
7   PRIMARY KEY (`D01F01`),
8   UNIQUE INDEX `D01F03_UNIQUE` (`D01F03` ASC) VISIBLE,
9   INDEX `OnDeptID_idx` (`T01F01` ASC) VISIBLE,
10  CONSTRAINT `OnDeptID`
11    FOREIGN KEY (`T01F01`)
12      REFERENCES `databasetraining`.`dept01` (`T01F01`)
13 )
14 COMMENT = 'StudentTable';
15

```

**Output:**

#	Time	Action	Message
✓ 1	14:53:42	CREATE TABLE `databasetraining`.`STD01` ( `D01F01` INT NOT NULL AUTO_INCREMENT COMMENT 'Student...	0 row(s) affected

## Table Constraints

### Primary Key

- A column that is set as a Primary Key must have unique values and contain NULL Value.
- Primary Key is used to uniquely identify Row from Table.

### Auto Increment

- Specifies that we don't have to insert data on the column, its value will be auto-added by incrementing value by 1 from previously inserted data.
- The data type must be in Integer and the default value is 1.

### Foreign Key

- When Data from a particular column refer to another table then we have to include foreign key contains into our table.

### Unique Index

- It specifies that this column must have unique data and it also can be NULL.

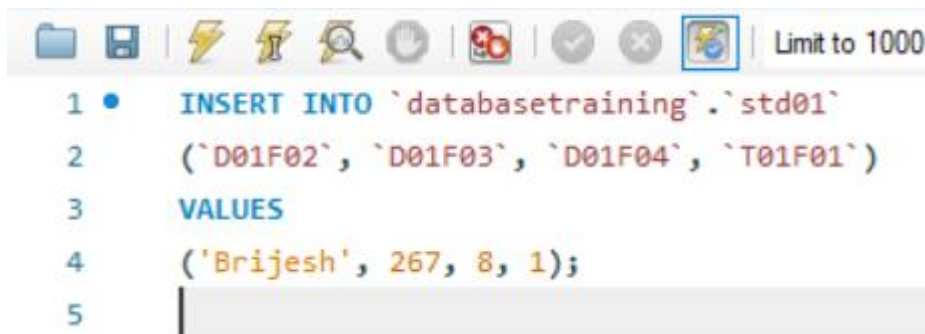
## Insert Command

- INSERT INTO is used to store data in the tables. The INSERT command creates a new row in the table to store data. The data is usually supplied by application programs that run on top of the database.

### Syntax:

```
INSERT INTO `table_name` (column_1, column_2, ...) VALUES (value_1, value_2, ...);
```

### Example:



### Output:

#	Time	Action	Message
✓ 1	15:44:17	INSERT INTO `databasetraining`.`std01` (`D01F02`, `D01F03`, `D01F04`, `T01F01`) VALUES (`Brijesh`, 267, 8, 1)	1 row(s) affected

## Select Command

- SELECT QUERY is used to fetch the data from the MySQL database.

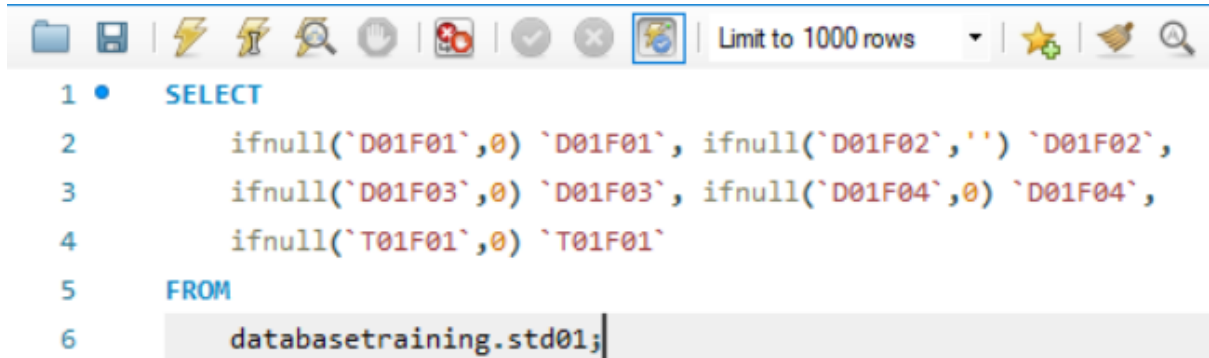
### Syntax:

```
SELECT [DISTINCT|ALL] { * | [fieldExpression [AS newName]] } FROM tableName
[alias] [WHERE condition] [GROUP BY fieldName(s)] [HAVING condition] ORDER
BY fieldName(s) | [Limit value]
```

- **DISTINCT**: Specify return rows must be Unique.
- **AS**: it is used to give an Alias to Table name or column name.
- **\***: used for returning all columns. (Not recommended).
- **WHERE**: it works like if condition in any programming language. This clause is used to compare the given value with the field value available in a MySQL table. If the given value from outside is equal to the available field value in the MySQL table, then it returns that row.
- **Group By**: it is used for grouping column data for applying the Aggregate function.
- **HAVING** condition is used to specify criteria when working using the GROUP BY keyword.

- **Order by** It is used for ordering data in Accessing or Descending.
- **Limit:** it is used for limiting the number of rows to return.

### Selecting All Data





```

1 • SELECT
2     ifnull(`D01F01`,0) `D01F01`, ifnull(`D01F02`,``) `D01F02`,
3     ifnull(`D01F03`,0) `D01F03`, ifnull(`D01F04`,0) `D01F04`,
4     ifnull(`T01F01`,0) `T01F01`
5 FROM
6     databasetraining.std01;

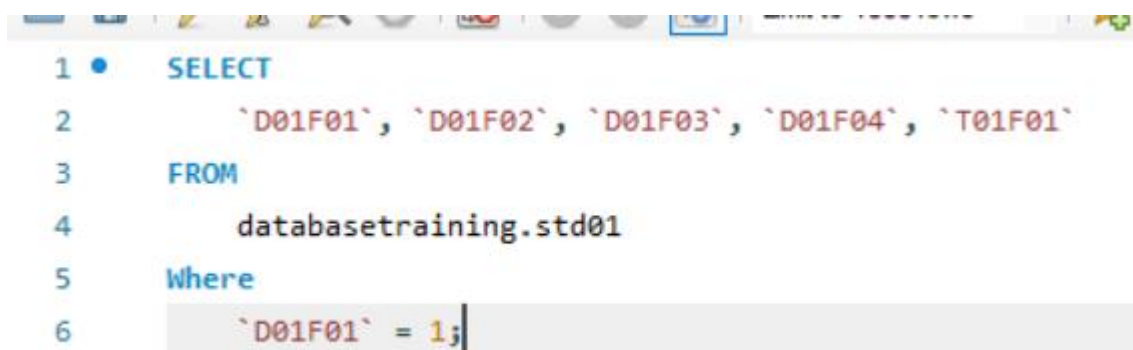
```

### Output:

Result Grid				Filter Rows:		Exp
	D01F01	D01F02	D01F03	D01F04	T01F01	
▶	1	Brijesh	267	8	1	
	2	Rajdeep	268	8	1	
	3	Karan	269	4	1	

- Here ifnull() function is used for providing default value if the column contains a null value.

### Where Clause



```

1 • SELECT
2     `D01F01`, `D01F02`, `D01F03`, `D01F04`, `T01F01`
3 FROM
4     databasetraining.std01
5 Where
6     `D01F01` = 1;

```

**Output:**

Result Grid					
	D01F01	D01F02	D01F03	D01F04	T01F01
▶	1	Brijesh	267	8	1
•	NULL	NULL	NULL	NULL	NULL

**WHERE clause combined with – AND LOGICAL Operator**

```

1 • SELECT
2     `D01F01`, `D01F02`, `D01F03`, `D01F04`, `T01F01`
3 FROM
4     databasetraining.std01
5 Where
6     `D01F01` = 1 OR `D01F01` = 2;

```

**Output:**

Result Grid					
	D01F01	D01F02	D01F03	D01F04	T01F01
▶	1	Brijesh	267	8	1
	2	Rajdeep	268	8	1
•	NULL	NULL	NULL	NULL	NULL

## Where with Wildcards Character

- MySQL Wildcards are characters that help search data matching complex criteria using Like, Not Like and Escape operators.

### Like

#### Example:

```

SELECT
    `D01F01`, `D01F02`, `D01F03`, `D01F04`, `T01F01`
FROM
    databasetraining.std01
Where
    `D01F02` like 'B%'
  
```

#### Output:

	D01F01	D01F02	D01F03	D01F04	T01F01
▶	1	Brijesh	267	8	1
•	NULL	NULL	NULL	NULL	NULL

- Here '%' is specified any 0 or more characters.

### Not Like

#### Example:

```

SELECT
    `D01F01`, `D01F02`, `D01F03`, `D01F04`, `T01F01`
FROM
    databasetraining.std01
Where
    `D01F03` not like '__7'
  
```

#### Output:

	D01F01	D01F02	D01F03	D01F04	T01F01
▶	2	Rajdeep	268	8	1
	3	Karan	269	4	1
•	NULL	NULL	NULL	NULL	NULL

- Here '\_' specify any one character.

## Group By

Finding the number of Students in a particular Semester using Group By

- **SELECT**

```
count(`D01F01`) `Group by D01F04`
FROM
    databasetraining.std01
Group By
    `D01F04`;
```

Output:

Result Grid		Filter
	Group by D01F04	
▶	2	
	1	

## Having Clause

Example:

- **SELECT**

```
count(`D01F01`) `Group by D01F04`
FROM
    databasetraining.std01
Group By
    `D01F04`
Having
    count('D01F01') > 1;
```

Output:

Result Grid		Filter
	Group by D01F04	
▶	2	



## Order By

Example:

```

• SELECT
    `D01F02`
  FROM
    databasetraining.std01
  Order By
    `D01F02` DESC;

```

Output:

	D01F02
▶	Rajdeep
	Karan
	Brijesh

## Update Query

- *It is used for Updating Existing Data.*

Syntax:

```

UPDATE table_name
SET column1 = value1, column2 = value2,
WHERE condition;

```

Example:

```

1 • UPDATE
2   `std01`
3   SET
4   `D01F03` = 261
5   WHERE
6   `D01F01` = 1

```

Checking if Value Change or not by selecting Query:

	D01F01	D01F02	D01F03	D01F04	T01F01
▶	1	Brijesh	261	8	1
	2	Rajdeep	268	8	1
	3	Karan	269	4	1
★	NULL	NULL	NULL	NULL	NULL

## Delete Query

- It Is Used for Deleting Existing Rows

Syntax:

```
DELETE FROM table_name WHERE condition;
```

Example:

```
1 DELETE from `std01` WHERE `D01F01` = 3;
```

Checking if Row is Deleted or not:

	D01F01	D01F02	D01F03	D01F04	T01F01
▶	1	Brijesh	261	8	1
	2	Rajdeep	268	8	1
★	NULL	NULL	NULL	NULL	NULL

## Aggregates Functions

AGGREGATE FUNCTION	DESCRIPTIONS
<b>COUNT()</b>	It returns the number of rows, including rows with NULL values in a group.
<b>SUM()</b>	It returns the total summed values (Non-NULL) in a set.
<b>AVERAGE()</b>	It returns the average value of an expression.
<b>MIN()</b>	It returns the minimum (lowest) value in a set.
<b>MAX()</b>	It returns the maximum (highest) value in a set.
<b>GROUP_CONCAT()</b>	It returns a concatenated string.
<b>JSON_ARRAYAGG()</b>	Return result set as a single JSON array
<b>JSON_OBJECTAGG()</b>	Return result set as a single JSON object
<b>STD()</b>	Return the population standard deviation
<b>VARIANCE()</b>	Return the population standard variance

### Examples:

# count() DEMO:

```
SELECT
    count(`D01F01`) AS 'NO. of Students'
FROM
    `std01`;
```

# OUTPUT:

```
#      NO. of Students
#      2
```

# sum() DEMO:

```
SELECT
    sum(`D01F01`) AS 'SUM of Students ID'
FROM
    `std01`;
```

# OUTPUT:

```
#      SUM of Students ID
#      3
```

# avg() Demo

```
SELECT
```

```
        avg(`D01F04`) AS 'Average value of Students  
Semester'  
FROM  
    `std01`;
```

# OUTPUT:

```
#      Avrage value of Students Semester  
#      8.0000
```

# min() Demo

```
SELECT  
    min(`D01F03`) AS 'Minimum value of Students  
EnrollmentNO'  
FROM  
    `std01`;
```

# OUTPUT:

```
#      Minimum value of Students EnrollmentNO  
#      261
```

# max() Demo

```
SELECT  
    max(`D01F03`) AS 'Maximum value of Students  
EnrollmentNO'  
FROM  
    `std01`;
```

# OUTPUT:

```
#      Maximum value of Students EnrollmentNO  
#      268
```

# group\_concat() Demo

```
SELECT  
    group_concat(`D01F02`) AS 'All Students Name'  
FROM  
    `std01`;
```

# OUTPUT:

```
#      All Students Name
```

```
#      Brijesh,Rajdeep

# JSON_ARRAYAGG() Demo

SELECT
    JSON_ARRAYAGG(`D01F02`) AS 'All Students Name'
FROM
    `STD01`;

# Output:
#      All Students Name
#      ["Brijesh", "Karan", "Rajdeep"]

# JSON_OBJECTAGG() demo

SELECT
    JSON_OBJECTAGG(`D01F01`, `D01F02`) AS 'All Students
With ID'
FROM
    `STD01`;

# Output:
#      All Students With ID
#      {"1": "Brijesh", "2": "Rajdeep", "3": "Karan"}

# STD() demo

SELECT
    STD(`D01F06`)
FROM
    `STD01`;

# OUTPUT:
#      STD(`D01F06`)
#      32.99831645537222

# variance() demo

SELECT
    variance(`D01F06`)
FROM
```

```
`STD01`;
```

```
# OUTPUT:
```

```
#      variance(`D01F06`)
```

```
#      1088.888888888889
```

## General

### Null Value & Keyword in MySQL

- **NULL** means “a missing unknown value” and it is treated somewhat differently from other values.
- To test for **NULL**, use the **IS NULL** and **IS NOT NULL** operators, as shown here:

```
SELECT
```

```
    `D01F01` IS NULL, `D01F01` IS NOT NULL
```

```
FROM
```

```
    `std01`
```

```
# OUTPUT
```

```
#    `D01F01` IS NULL,    `D01F01` IS NOT NULL
```

```
#    0,                    1
```

```
#    0,                    1
```

- Here, 0 means False and 1 means True;
- So If Value is NULL then IS NULL returns 1 or if NOT NULL then It Returns 0.
- Similarly, IF VALUE Is NULL Then IS NOT NULL Returns 0 or If NOT NULL then it returns 1.

### Auto Increment

- Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table.
- Often this is the primary key field that we would like to be created automatically every time a new record is inserted.

For Example:

```
CREATE TABLE ANML01 (
    L01F01 INT NOT NULL AUTO_INCREMENT,
    L01F02 CHAR(30) NOT NULL,
    PRIMARY KEY (`L01F01`)
);
```

```
INSERT INTO ANML01 (`L01F02`) VALUES
    ('dog'),('cat'),('penguin'),
    ('lax'),('whale'),('ostrich');
```

```
SELECT
    `L01F01`, `L01F02`
FROM
    `ANML01`;
```

```
# OUTPUT:
#  L01F01, L01F02
#  1, dog
#  2, cat
#  3, penguin
#  4, lax
#  5, whale
#  6, ostrich
```

- No value was specified for the **AUTO\_INCREMENT** column, so MySQL assigned sequence numbers automatically. You can also explicitly assign 0 to the column to generate sequence numbers unless the **NO\_AUTO\_VALUE\_ON\_ZERO** SQL mode is enabled.



- If the column is declared **NOT NULL**, it is also possible to assign **NULL** to the column to generate sequence numbers.
- When you insert any other value into an **AUTO\_INCREMENT** column, the column is set to that value and the sequence is reset so that the next automatically generated value follows sequentially from the largest column value.

For example:

```
INSERT INTO `ANML01` (`L01F01`,`L01F02`) VALUES(0,'groundhog');
INSERT INTO `ANML01` (`L01F01`,`L01F02`) VALUES(NULL,'squirrel');
INSERT INTO `ANML01` (`L01F01`,`L01F02`) VALUES(100,'rabbit');
INSERT INTO `ANML01` (`L01F01`,`L01F02`) VALUES(NULL,'mouse');
SELECT
    `L01F01`, `L01F02`
FROM
    `ANML01`;
```

# OUTPUT:

```
# | L01F01, L01F02
# 1, dog
# 2, cat
# 3, penguin
# 4, lax
# 5, whale
# 6, ostrich
# 7, groundhog
# 8, squirrel
# 100, rabbit
# 101, mouse
```

- Updating an existing **AUTO\_INCREMENT** column value also resets the **AUTO\_INCREMENT** sequence.
- You can retrieve the most recent automatically generated **AUTO\_INCREMENT** value with the **LAST\_INSERT\_ID()** SQL function or the **mysql\_insert\_id()** C API function. These functions are connection-specific, so their return values are not affected by another connection that is also performing inserts.

- To start with an AUTO\_INCREMENT value other than 1, set that value with CREATE TABLE or ALTER TABLE, like this:

```
ALTER TABLE `ANML01` AUTO_INCREMENT = 500;  
INSERT INTO `ANML01` (`L01F02`) VALUES('cow');  
SELECT  
    `L01F01`, `L01F02`  
FROM  
    `ANML01`;
```

# OUTPUT:

```
# L01F01, L01F02  
# 1, dog  
# 2, cat  
# 3, penguin  
# 4, lax  
# 5, whale  
# 6, ostrich  
# 7, groundhog  
# 8, squirrel  
# 100, rabbit  
# 101, mouse  
# 500, cow
```

## Alter, Drop & rename

### ALTER Command

- MySQL ALTER statement is used when you want to change the name of your table or any table field. It is also used to add or delete an existing column in a table.

#### 1) ADD a column in the table:

#### Syntax:

```
ALTER TABLE table_name
ADD new_column_name column_definition
[ FIRST | AFTER column_name ];
```

#### Parameters:

- table\_name:** It specifies the name of the table that you want to modify.
- new\_column\_name:** It specifies the name of the new column that you want to add to the table.
- column\_definition:** It specifies the data type and definition of the column (NULL or NOT NULL, etc).
- FIRST | AFTER column\_name:** It is optional. It tells MySQL where in the table to create the column. If this parameter is not specified, the new column will be added to the end of the table.

#### Example:

```
ALTER TABLE
  `STD01`
ADD
  `D01F05` varchar(5) NOT NULL COMMENT 'Student Grade';

SELECT
  `D01F05`
FROM
  `STD01`;

# OUTPUT:
#   D01F05
#
#
```

## 2) MODIFY column in the table:

- The MODIFY command is used to change the column definition of the table.

### Syntax:

```
ALTER TABLE table_name  
MODIFY column_name column_definition  
[ FIRST | AFTER column_name ];
```

### Example:

```
ALTER TABLE  
    `STD01`  
MODIFY  
    `D01F05` varchar(3) NULL COMMENT 'StudentGrade';
```

```
SELECT  
    `D01F05`  
FROM  
    `STD01`;
```

```
# OUTPUT:  
#   D01F05  
#  
#
```

### Output:

16:35:36 ALTER TABLE 'STD01' MODIFY 'D01F05' varchar(3) NULL COMMENT 'StudentGrade'

2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0

**Showing Columns:**

```
SHOW COLUMNS FROM `STD01`;
```

**Output:**

Field	Type	Null	Key	Default	Extra
D01F01	int	NO	PRI	NULL	auto_increment
D01F02	varchar(100)	NO		NULL	
D01F03	int	NO	UNI	NULL	
D01F04	tinyint	NO		NULL	
T01F01	int	NO	MUL	NULL	
D01F05	varchar(3)	YES		NULL	

**3) CHANGE Column Name:**

- WE can change the name as well as also Modify Column In This way:

**Example:**

```
ALTER TABLE `STD01` CHANGE COLUMN `D01F05` `D01F06`  
VARCHAR(3) NULL COMMENT 'StudentGrade';  
ALTER TABLE `STD01` CHANGE COLUMN `T01F01` `D01F05` INT NOT  
NULL COMMENT 'DeptID';  
SHOW COLUMNS FROM `STD01`;
```

**Output:**

Field	Type	Null	Key	Default	Extra
D01F01	int	NO	PRI	NULL	auto_increment
D01F02	varchar(100)	NO		NULL	
D01F03	int	NO	UNI	NULL	
D01F04	tinyint	NO		NULL	
D01F05	int	NO	MUL	NULL	
D01F06	varchar(3)	YES		NULL	

#### 4) DROP Column

**Example:**

```
ALTER TABLE `STD01` DROP COLUMN `D01F06` ;  
SHOW COLUMNS FROM `STD01` ;
```

**Output:**

Field	Type	Null	Key	Default	Extra
D01F01	int	NO	PRI	NULL	auto_increment
D01F02	varchar(100)	NO		NULL	
D01F03	int	NO	UNI	NULL	
D01F04	tinyint	NO		NULL	
D01F05	int	NO	MUL	NULL	

## RENAME Command

- Sometimes our table name is non-meaningful, so it is required to rename or change the name of the table. MySQL provides a useful syntax that can rename one or more tables in the current database.

### 1) Rename Table

#### Syntax:

```
RENAME old_table TO new_table;
```

#### Example:

```
RENAME TABLE `ANML01` TO `ANM01`;  
SELECT `L01F01` FROM `ANM01`;
```

#### Output:

L01F01
4
5
6
7
8
100
101
500
NULL

## DROP Command

- DROP Command Is used to DELETE TABLE;

### Syntax:

```
DROP TABLE `TABLE_NAME` ;
```

### Example:

```
DROP TABLE `ANM01` ;
```

### Output:

#	Time	Action	Message
✓ 1	13:38:47	DROP TABLE 'ANM01'	0 row(s) affected



## LIMIT keyword

- The LIMIT clause is used in the SELECT statement to constrain the number of rows to return. The LIMIT clause accepts one or two arguments. The values of both arguments must be zero or positive integers.

### Syntax:

```
SELECT
    select_list
FROM
    table_name
LIMIT [offset,] row_count;
```

- The offset specifies the offset of the first row to return. The offset of the first row is 0, not 1.
- The row\_count specifies the maximum number of rows to return.

### Example:

```
SELECT
    `D01F02`
FROM
    `STD01`
LIMIT
    1;
```

```
# OUTPUT:
#      D01F02
#      Brijesh
```

```
SELECT
    `D01F02`
FROM
    `STD01`
LIMIT
    1,1;
```

```
# OUTPUT:
#      D01F02
#      Rajdeep
```

## Sub-Queries

- A subquery in MySQL is a query, which is nested into another SQL query and embedded with SELECT, INSERT, UPDATE or DELETE statement along with the various operators.
- We can also nest the subquery with another subquery.
- A subquery is known as the **inner query**, and the query that contains the subquery is known as the **outer query**.
- The inner query executed first gives the result to the outer query, and then the main/outer query will be performed.

The following are the rules to use subqueries:

- Subqueries should always use in **parentheses**.
- If the main query does not have multiple columns for subquery, then a subquery can have only one column in the SELECT command.
- We can use various comparison operators with the subquery, such as >, <, =, IN, ANY, SOME, and ALL. A multiple-row operator is very useful when the subquery returns more than one row.
- We cannot use the **ORDER BY** clause in a subquery, although it can be used inside the main query.

**Syntax:**

```
SELECT column_list (s) FROM table_name
WHERE column_name OPERATOR
      (SELECT column_list (s) FROM table_name [WHERE])
```

### 1) MySQL Subquery with IN Operator

```
SELECT
  `D01F01`, `D01F02`, `D01F03`
FROM
  `STD01`
WHERE
  `D01F01`
IN
  (
    SELECT
      `D01F01`
    FROM
      `STD01`
    WHERE
      `D01F03` > 261
  )
;
```

**Output:**

D01F01	D01F02	D01F03
2	Rajdeep	268
NULL	NULL	NULL

## 2) MySQL Subquery with NOT-IN Operator

```

SELECT
    `D01F01`, `D01F02`, `D01F03`
FROM
    `STD01`
WHERE
    `D01F01`
NOT IN
    (
        SELECT
            `D01F01`
        FROM
            `STD01`
        WHERE
            `D01F03` > 261
    )
;

```

### Output:

D01F01	D01F02	D01F03
1	Brijesh	261
NULL	NULL	NULL

## 3) MySQL Subquery in the FROM Clause

```

SELECT
    Max(`D01F01`), MIN(`D01F01`), FLOOR(AVG(`D01F01`))
FROM
    (
        SELECT
            `D01F01`, COUNT(`D01F01`) AS TotalStudents
        FROM
            `STD01`
        GROUP BY
            `D01F05`
    )
AS
    Student_Group_By_Department
;

```

### Output:

Max(`D01F01`)	MIN(`D01F01`)	FLOOR(AVG(`D01F01`))
1	1	1

#### 4) MySQL Correlated Subqueries

```

SELECT
    `D01F02`, `D01F03`, `D01F04`
FROM
    `STD01`
WHERE
    `D01F05` >= ( SELECT
                    `T01F01`
                  FROM
                    `DEPT01`
                  WHERE
                    `T01F01` = `D01F05`
                  )
;

```

#### Output:

D01F02	D01F03	D01F04
Brijesh	261	8
Rajdeep	268	8

#### 5) MySQL Subqueries with EXISTS

```

SELECT
    `D01F02`, `D01F03`, `D01F04`
FROM
    `STD01`
WHERE
    EXISTS (      SELECT
                  `T01F01`
                FROM
                  `DEPT01`
                WHERE
                  `T01F01` = `D01F05`
                )
;

```

#### Output:

D01F02	D01F03	D01F04
Brijesh	261	8
Rajdeep	268	8

## 6) MySQL Subqueries with NOT EXISTS

```
SELECT
    `D01F02`, `D01F03`, `D01F04`
FROM
    `STD01`
WHERE
    NOT EXISTS (
        SELECT
            `T01F01`
        FROM
            `DEPT01`
        WHERE
            `T01F01` = `D01F05`
    )
;
```

**Output:**

D01F02	D01F03	D01F04
--------	--------	--------

## Joins

- MySQL JOINS are used with SELECT statements. It is used to retrieve data from multiple tables. It is performed whenever you need to fetch records from two or more tables.

There are three types of MySQL joins:

- MySQL INNER JOIN (or sometimes called a simple join)
- MySQL LEFT OUTER JOIN (or sometimes called LEFT JOIN)
- MySQL RIGHT OUTER JOIN (or sometimes called RIGHT JOIN)

## MySQL Inner JOIN (Simple Join)

- The MySQL INNER JOIN is used to return all rows from multiple tables where the join condition is satisfied. It is the most common type of join.

**Syntax:**

```
SELECT columns
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

**Example:**

```

SELECT
    `D01F02`, `D01F03`, `D01F04`, `T01F02`
FROM
    `STD01`
INNER JOIN
    `DEPT01`
ON
    `D01F05` = `T01F01`;

```

**Output:**

D01F02	D01F03	D01F04	T01F02
Brijesh	261	8	Computer
Rajdeep	268	8	Computer
Karan	264	8	Civil

## MySQL Left Outer Join

- The LEFT OUTER JOIN returns all rows from the left-hand table specified in the ON condition and only those rows from the other table where the join condition is fulfilled.

**Syntax:**

```

SELECT columns
FROM table1
LEFT [OUTER] JOIN table2
ON table1.column = table2.column;

```

**Example:**

```

SELECT
    `D01F02`, `D01F03`, `D01F04`, `T01F02`
FROM
    `STD01`
LEFT JOIN
    `DEPT01`
ON
    `D01F05` = `T01F01`;

```

**Output:**

D01F02	D01F03	D01F04	T01F02
Brijesh	261	8	Computer
Rajdeep	268	8	Computer
Karan	264	8	Civil

## MySQL Right Outer Join

- The MySQL Right Outer Join returns all rows from the RIGHT-hand table specified in the ON condition and only those rows from the other table where the join condition is fulfilled.

### Syntax:

```
SELECT columns
FROM table1
RIGHT [OUTER] JOIN table2
ON table1.column = table2.column;
```

### Example:

```
SELECT
    `D01F02`, `D01F03`, `D01F04`, `T01F02`
FROM
    `STD01`
RIGHT JOIN
    `DEPT01`
ON
    `D01F05` = `T01F01`;
```

### Output:

D01F02	D01F03	D01F04	T01F02
Karan	264	8	Civil
Rajdeep	268	8	Computer
Brijesh	261	8	Computer

## Unions

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

- Every SELECT statement within UNION must have the same number of columns
- The columns must also have similar data types
- The columns in every SELECT statement must also be in the same order

### Syntax

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```

**Example:**

```

SELECT
    `T01F02`
FROM
    `DEPT01`
UNION
SELECT
    `T02F02`
FROM
    `DEPT02`;

```

**Output:**

T01F02
Civil
Computer

**UNION ALL**

- The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL:

**Syntax:**

```

SELECT column_name(s) FROM table1 UNION ALL
SELECT column_name(s) FROM table2;

```

- Note:** The column names in the result-set are usually equal to the column names in the first SELECT statement.

**Example:**

```

SELECT
    `T01F02`
FROM
    `DEPT01`
UNION ALL
SELECT
    `T02F02`
FROM
    `DEPT02`;

```



**Output:**

T01F02
Civil
Computer
Civil
Computer

## Views

- A view is a database object that has no values.
- Its contents are based on the base table.
- It contains rows and columns similar to the real table.
- In MySQL, the View is a **virtual table** created by a query by joining one or more tables.
- It is operated similarly to the base table but does not contain any data of its own.
- The View and table have one main difference the views are definitions built on top of other tables (or views).
- If any changes occur in the underlying table, the same changes are reflected in the View also.

**Syntax:**

```
CREATE [OR REPLACE] VIEW view_name AS
SELECT columns
FROM tables
[WHERE conditions];
```

**Parameters:**

- **OR REPLACE:** It is optional. It is used when a VIEW already exists. If you do not specify this clause and the VIEW already exists, the CREATE VIEW statement will return an error.
- **view\_name:** It specifies the name of the VIEW that you want to create in MySQL.
- **WHERE conditions:** It is also optional. It specifies the conditions that must be met for the records to be included in the VIEW.

**Example:**

```
CREATE VIEW
  `vv_STD01`
AS
  SELECT
    `D01F01`, `D01F02`
  FROM
    `STD01`
;

SELECT
  `D01F02`
FROM
  `vv_STD01`;
```

**Output:**

D01F02
Brijesh
Rajdeep
Karan

## Index

- An index is a data structure that allows us to add indexes to the existing table.
- It enables you to improve the faster retrieval of records on a database table.
- It creates an **entry** for each value of the indexed columns.
- We use it to quickly find the record without searching each row in a database table whenever the table is accessed.
- We can create an index by using one or more **columns** of the table for efficient access to the records.
- When a table is created with a primary key or unique key, it automatically creates a special index named **PRIMARY**.
- We called this index a clustered index.
- All indexes other than PRIMARY indexes are known as non-clustered indexes or secondary indexes.

### Syntax:

**CREATE INDEX** [index\_name] **ON** [table\_name] (**column** names)

### Example:

```
CREATE INDEX `idx_D01F02` ON `STD01`(`D01F02`);
```

### Output:

✓	54	13:24:40	CREATE INDEX `idx_D01F02` ON `STD01`(`D01F02`)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0
---	----	----------	--	--

## CASE-WHEN-THEN

- The CASE statement goes through conditions and returns a value when the first condition is met (like an IF-THEN-ELSE statement). So, once a condition is true, it will stop reading and return the result.
- If no conditions are true, it will return the value in the ELSE clause.
- If there is no ELSE part and no conditions are true, it returns NULL.

### Syntax:

```
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  WHEN conditionN THEN resultN
  ELSE result
END;
```

### Parameters:

Parameter	Description
<i>condition1, condition2, ...conditionN</i>	Required. The conditions. These are evaluated in the same order as they are listed
<i>result1, result2, ...resultN</i>	Required. The value to return once a condition is true

### Example:

```
SELECT
  `D01F01` AS 'STUDENT ID',
  `D01F02` AS 'STUDENT NAME',
  CASE
    WHEN `D01F06` > 85 AND `D01F06` <= 100 THEN 'Distinction'
    WHEN `D01F06` > 60 AND `D01F06` <= 85 THEN 'FIRST CLASS'
    WHEN `D01F06` > 45 AND `D01F06` <= 60 THEN 'SECOND CLASS'
    WHEN `D01F06` > 33 AND `D01F06` <= 45 THEN 'THIRD CLASS'
    WHEN `D01F06` > 0 AND `D01F06` <= 33 THEN 'FAIL'
    ELSE 'Invalid'
  END as 'Student Class'
FROM
  `STD01`;
```

**Output:**

STUDENT ID	STUDENT NAME	Student Class
1	Brijesh	Distinction
2	Rajdeep	SECOND CLASS
3	Karan	FAIL

**Stored-Procedures**

- A procedure (often called a stored procedure) is a **collection of pre-compiled SQL statements** stored inside the database.
- It is a subroutine or a subprogram in the regular computing language.
- **A procedure always contains a name, parameter lists, and SQL statements.**
- We can invoke the procedures by using triggers, other procedures and other applications.

**Syntax:**

```

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

```

**Parameters:**

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.

## CALL PROCEDURE:

- We can use the **CALL statement** to call a stored procedure. This statement returns the values to its caller through its parameters (IN, OUT, or INOUT).

### Syntax:

```
CALL procedure_name ( parameter(s))
```

### Example:

```
DELIMITER &&
CREATE PROCEDURE sp_FindGradeFromStudnetMarks (IN p_StudentID INT)
BEGIN
    SELECT
        CASE
            WHEN `D01F06` > 85 AND `D01F06` <= 100 THEN 'Distinction'
            WHEN `D01F06` > 60 AND `D01F06` <= 85 THEN 'FIRST CLASS'
            WHEN `D01F06` > 45 AND `D01F06` <= 60 THEN 'SECOND CLASS'
            WHEN `D01F06` > 33 AND `D01F06` <= 45 THEN 'THIRD CLASS'
            WHEN `D01F06` > 0 AND `D01F06` <= 33 THEN 'FAIL'
            ELSE 'Invalid'
        END as 'Student Class'
    FROM
        `STD01`
    WHERE
        `D01F01` = p_StudentID;
END &&
DELIMITER ;

CALL sp_FindGradeFromStudnetMarks(1);
```

**Output:-**

Student Class
Distinction

**Functions**

- A stored function in MySQL is a set of SQL statements that perform some task/operation and return a single value.
- It is one of the types of stored programs in MySQL.
- When you will create a stored function, make sure that you have a CREATE ROUTINE database privilege.
- Generally, we used this function to encapsulate the common business rules or formulas reusable in stored programs or SQL statements.

The stored function is almost similar to the procedure in MySQL, but it has some differences that are as follows:

- The function parameter may contain only the **IN parameter** but can't allow specifying this parameter, while the procedure can allow **IN, OUT, INOUT parameters**.
- The stored function can return only a single value defined in the function header.
- The stored function may also be called within SQL statements.
- It may not produce a result set.

**Syntax:**

```
DELIMITER $$

CREATE FUNCTION fun_name(fun_parameter(s))
RETURNS datatype
[NOT] {Characteristics}
fun_body;
```

**Parameters:**

Parameter Name	Descriptions
fun_name	It is the name of the stored function that we want to create in a database
fun_parameter	It contains the list of parameters used by the function body. It does not allow to specify IN, OUT, INOUT parameters.
datatype	It is a data type of return value of the function. It should any valid MySQL data type.
characteristics	The CREATE FUNCTION statement only accepted when the characteristics (DETERMINISTIC, NO SQL, or READS SQL DATA) are defined in the declaration.
fun_body	This parameter has a set of SQL statements to perform the operations. It requires at least one RETURN statement. When the return statement is executed, the function will be terminated automatically. The function body is given below: BEGIN -- SQL statements END \$\$ DELIMITER



**Example:**

```

DELIMITER $$
CREATE FUNCTION
    fu_FindGradeOfStudent (p_StudentID int)
RETURNS
    VARCHAR(20)
DETERMINISTIC
BEGIN
    DECLARE v_StudentGrade VARCHAR(20);
    DECLARE v_Marks DECIMAL(5,2);

    SELECT
        `D01F06` INTO v_Marks
    FROM
        `STD01`
    WHERE
        `D01F01` = p_StudentID;

    IF
        v_Marks > 85 AND v_Marks <= 100
    THEN
        SET v_StudentGrade = 'Distinction';
    ELSEIF
        v_Marks > 60 AND v_Marks <= 85
    THEN
        SET v_StudentGrade = 'FIRST CLASS';
    ELSEIF
        v_Marks > 45 AND v_Marks <= 60
    THEN
        SET v_StudentGrade = 'SECOND CLASS';
    ELSEIF
        v_Marks > 33 AND v_Marks <= 45
    THEN
        SET v_StudentGrade = 'THIRD CLASS';
    ELSEIF
        v_Marks > 0 AND v_Marks <= 33
    THEN
        SET v_StudentGrade = 'FAIL';
    ELSE
        SET v_StudentGrade = 'Error: Something Wrong';
    END IF;

    RETURN (v_StudentGrade);
END$$
DELIMITER ;

SELECT
    `fu_FindGradeOfStudent`(2);

```

**Output:**

`fu_FindGradeOfStudent`(2)
SECOND CLASS

# Advance

## Partitions

- Partitioning in MySQL is used to split or partition the rows of a table into separate tables in different locations, but still, it is treated as a single table.
- It distributes the portions of the table's data across a file system based on the rules we have set as our requirement.

MySQL has mainly two forms of partitioning:

### 1. Horizontal Partitioning

- This partitioning split the rows of a table into multiple tables based on our logic. In horizontal partitioning, the number of columns is the same in each table, but no need to keep the same number of rows. It physically divides the table but logically treated as a whole. Currently, MySQL supports this partitioning only.

### 2. Vertical Partitioning

- This partitioning splits the table into multiple tables with fewer columns from the original table. It uses an additional table to store the remaining columns.

### Types of MySQL Partitioning

MySQL has mainly six types of partitioning, which are given below:

1. RANGE Partitioning
2. LIST Partitioning
3. COLUMNS Partitioning
4. HASH Partitioning
5. KEY Partitioning
6. Subpartitioning

## RANGE Partitioning

- This partitioning allows us to partition the rows of a table based on column values that fall within a specified range.
- The given range is always in a contiguous form but should not overlap each other, and also uses the **VALUES LESS THAN** operator to define the ranges.

**Example:**

```
CREATE TABLE `DEPT03` (
  `T03F01` INT NOT NULL PRIMARY KEY COMMENT 'DepartmentID',
  `T03F02` VARCHAR(40) NOT NULL
)
PARTITION BY RANGE(`T03F01`)
PARTITIONS 3
( PARTITION part0 VALUES LESS THAN (20),
  PARTITION part1 VALUES LESS THAN (40),
  PARTITION part2 VALUES LESS THAN (60)) ;
```

- We can see the partition created by CREATE TABLE statement using the below query:

```
SELECT
  TABLE_NAME, PARTITION_NAME, TABLE_ROWS, AVG_ROW_LENGTH, DATA_LENGTH
FROM
  INFORMATION_SCHEMA.PARTITIONS
WHERE
  TABLE_SCHEMA = 'databasetraining' AND TABLE_NAME = 'DEPT03';
```

**Output:**

TABLE_NAME	PARTITION_NAME	TABLE_ROWS	AVG_ROW_LENGTH	DATA_LENGTH
dept03	part0	1	16384	16384
dept03	part1	3	5461	16384
dept03	part2	1	16384	16384

**DROP MySQL Partition**

```
ALTER TABLE `DEPT03` TRUNCATE PARTITION part0;
```

## LIST Partitioning

- It is the same as Range Partitioning.
- Here, the partition is defined and selected based on columns matching one of a set of discrete value lists rather than a set of a contiguous range of values.

### Example:

```
CREATE TABLE `DEPT04` (
  `T04F01` INT NOT NULL PRIMARY KEY COMMENT 'DepartmentID',
  `T04F02` VARCHAR(40) NOT NULL
)
PARTITION BY LIST(`T04F01`)
( PARTITION part0 VALUES IN (20,30),
  PARTITION part1 VALUES IN (40,50),
  PARTITION part2 VALUES IN (60,70)) ;
```

## HASH Partitioning

- This partitioning is used to distribute data based on a **predefined number** of partitions.
- In other words, it splits the table as of the value returned by the user-defined expression.

### Example:

```
CREATE TABLE `DEPT05` (
  `T05F01` INT NOT NULL PRIMARY KEY COMMENT 'DepartmentID',
  `T05F02` VARCHAR(40) NOT NULL
)
PARTITION BY HASH(`T05F01`)
PARTITION 2;
```

## KEY Partitioning

- It is similar to the HASH partitioning where the hash partitioning uses the user-specified expression, and MySQL server supplied the hashing function for key.

### Example:

```
CREATE TABLE `DEPT06` (
  `T06F01` INT NOT NULL PRIMARY KEY COMMENT 'DepartmentID',
  `T06F02` VARCHAR(40) NOT NULL
)
PARTITION BY KEY()
PARTITION 2;
```

## Range Column Partitioning

- It is similar to the range partitioning with one difference. It defines partitions using ranges based on various columns as partition keys.
  - The defined ranges are of column types other than an integer type.

### Example:

```
CREATE TABLE `DEPT07` (
  `T07F01` INT NOT NULL COMMENT 'DepartmentID',
  `T07F02` VARCHAR(40) NOT NULL
)
PARTITION BY RANGE COLUMNS(`T07F01`,`T07F02`)
( PARTITION part0 VALUES LESS THAN (20,'Computer'),
  PARTITION part1 VALUES LESS THAN (MAXVALUE,MAXVALUE));
```

## List Columns Partitioning

- It takes a list of single or multiple columns as partition keys.
- It enables us to use various columns of types other than integer types as partitioning columns.
- In this partitioning, we can use String data types, DATE, and DATETIME columns.

### Example:

```
CREATE TABLE `DEPT08` (
  `T08F01` INT NOT NULL COMMENT 'DepartmentID',
  `T08F02` VARCHAR(40) NOT NULL
)
PARTITION BY LIST COLUMNS (`T08F01`,`T08F02`)
( PARTITION part0 VALUES IN (20,'Computer'),
  PARTITION part1 VALUES IN (40,'Civil'));
```

## SUBPARTITIONING

- It is a composite partitioning that further splits each partition in a partition table. The below example helps us to understand it more clearly:

### Example:

```
CREATE TABLE `DEPT09` (
  `T09F01` INT NOT NULL PRIMARY KEY COMMENT 'DepartmentID',
  `T09F02` VARCHAR(40) NOT NULL
)
PARTITION BY RANGE(`T09F01`)
SUBPARTITION BY HASH(`T09F01`)
( PARTITION part0 VALUES LESS THAN (20),
  PARTITION part1 VALUES LESS THAN (40),
  PARTITION part2 VALUES LESS THAN (60)) ;
```

## Backup

### Command Syntax:

```
mysqldump -u [user name] -p [password] [options] [database_name]  
[tablename] > [dumpfilename.sql]
```

### Example:

```
mysqldump --user root --password --all-databases > all-databases.sql
```

### Restore Example:

```
mysql --user root --password mysql < all-databases.sql
```

To dump a specific database, use the name of the database instead of the `--all-database` parameter.

```
mysql --user root --password [db_name] < [db_name].sql
```