	Description	Example
*	The **asterisk (*)** metacharacter is used to match zero (0) or more instances of the strings preceding it	*SELECT * FROM movies WHERE title REGEXP 'da*';* will give all movies containing characters "da" .For Example, Da Vinci Code , Daddy's Little Girls.
+	The **plus (+)** metacharacter is used to match one or more instances of strings preceding it.	*SELECT * FROM `movies` WHERE `title` REGEXP 'mon+';* will give all movies containing characters "mon" .For Example, Angels and Demons.
?	The question**(?)**metacharacter is used to match zero (0) or one instances of the strings preceding it.	*SELECT * FROM `categories` WHERE `category_name` REGEXP 'com?';* will give all the categories containing string com .For Example, comedy , romantic comedy .
.	The **dot (.)** metacharacter is used to match any single character in exception of a new line.	*SELECT * FROM movies WHERE `year_released` REGEXP '200.';* will give all the movies released in the years starting with characters "200" followed by any single character .For Example, 2005,2007,2008 etc.
[abc]	The **charlist [abc]** is used to match any of the enclosed characters.	*SELECT * FROM `movies` WHERE `title` REGEXP '[vwxyz]';* will give all the movies containing any single character in "vwxyz" .For Example, X-Men, Da Vinci Code, etc.
[^abc]	The **charlist [^abc]** is used to match any characters excluding the ones enclosed.	*SELECT * FROM `movies` WHERE `title` REGEXP '^[^vwxyz]';* will give all the movies containing characters other than the ones in "vwxyz".

[A-Z]	The **[A-Z]** is used to match any upper case letter.	*SELECT * FROM `members` WHERE `postal_address` REGEXP '[A-Z];* will give all the members that have postal address containing any character from A to Z. .For Example, Janet Jones with membership number 1.
[a-z]	The **[a-z]** is used to match any lower case letter	*SELECT * FROM `members` WHERE `postal_address` REGEXP '[a-z];* will give all the members that have postal addresses containing any character from a to z. .For Example, Janet Jones with membership number 1.
[0-9]	The **[0-9]** is used to match any digit from 0 through to 9.	*SELECT * FROM `members` WHERE `contact_number` REGEXP '[0-9]'* will give all the members have submitted contact numbers containing characters "[0-9]" .For Example, Robert Phil.
^	The **caret (^)** is used to start the match at beginning.	*SELECT * FROM `movies` WHERE `title` REGEXP '^[cd];* gives all the movies with the title starting with any of the characters in "cd" .For Example, Code Name Black, Daddy's Little Girls and Da Vinci Code.
|	The **vertical bar (|)** is used to isolate alternatives.	*SELECT * FROM `movies` WHERE `title` REGEXP '^[cd]|^[u];* gives all the movies with the title starting with any of the characters in "cd" or "u" .For Example, Code Name Black, Daddy's Little Girl, Da Vinci Code and Underworld - Awakening.

REGEXP examples

SELECT * FROM `movies` WHERE `title` REGEXP 'code';

```
SELECT * FROM `movies` WHERE title REGEXP '^[abd
c]';
```

Functions	Description
ADDDATE()	MySQL ADDDATE() adds a time value with a date.
ADDTIME()	In MySQL the ADDTIME() returns a time or datetime after adding a time value with a time or datetime.
CURDATE()	In MySQL the CURDATE() returns the current date in 'YYYY-MM-DD' format or 'YYYYMMDD' format depending on whether numeric or string is used in the function.
CURRENT_DATE()	In MySQL the CURRENT_DATE returns the current date in 'YYYY-MM-DD' format or YYYYMMDD format depending on whether numeric or string is used in the function.
CURRENT_TIME()	In MySQL the CURRENT_TIME() returns the current time in 'HH:MM:SS' format or HHMMSS.uuuuuu format depending on whether numeric or string is used in the function.
CURRENT_TIMESTAMP()	In MySQL the CURRENT_TIEMSTAMP returns the current date and time in 'YYYY-MM-DD HH:MM:SS' format or YYYYMMDDHHMMSS.uuuuuu format depending on whether numeric or string is used in the function.
DATE_ADD()	MySQL DATE_ADD() adds time values (as intervals) to a date value.

	The ADDDATE() is the synonym of DATE_ADD().
DATE_FORMAT()	**MySQL DATE_FORMAT() formats a date as specified in the argument.**
DATE()	**MySQL DATE() takes the date part out from a datetime expression.**
DATEDIFF()	**MySQL DATEDIFF() returns the number of days between two dates or datetimes.**
DAY()	**MySQL DAY() returns the day of the month for a specified date.**
DAYNAME()	**MySQL DAYNAME() returns the name of the week day of a date specified in the argument.**
DAY OF MONTH()	**MySQL DAYOFMONTH() returns the day of the month for a given date.**
HOUR()	**MySQL HOUR() returns the hour of a time.**
LAST_DAY()	**MySQL LAST_DAY() returns the last day of the corresponding month for a date or datetime value.**

MONTH()	**MySQL MONTH() returns the month for the date within a range of 1 to 12 (January to December).**
MONTHNAME()	**MySQL MONTHNAME() returns the full name of the month for a given date.**
NOW()	**MySQL NOW() returns the value of current date and time in 'YYYY-MM-DD HH:MM:SS' format or YYYYMMDDHHMMSS.uuuuuu format depending on the context (numeric or string) of the function.**
TIME()	**MySQL TIME() extracts the time part of a time or datetime expression as string format.**
TIMEDIFF()	**MySQL TIMEDIFF() returns the differences between two time or datetime expressions.**
TIMESTAMP()	**MySQL TIMESTAMP() returns a datetime value against a date or datetime expression.**
TIMESTAMPADD()	**MySQL TIMESTAMPADD() adds time value with a date or datetime value.**
TIMESTAMPDIFF()	**MySQL the TIMESTAMPDIFF() returns a value after subtracting a datetime expression from another.**

TO_DAYS()	**MySQL TO_DAYS() returns number of days between a given date and year 0.**
YEAR()	**MySQL YEAR() returns the year for a given date.**
YEARWEEK()	**MySQL YEARWEEK() returns year and week number for a given date.**

Function	Description
ASCII	Returns the number code that represents the specific character
CONCAT	Concatenates two or more expressions together
CONCAT_WS	Concatenates two or more expressions together and adds a separator between them
FIELD	Returns the position of a value in a list of values
FIND_IN_SET	Returns the position of a string in a string list SELECT FIND_IN_SET('y','x,y,z'); -- 2 SELECT FIND_IN_SET('a','x,y,z'); ----0
FORMAT	Formats a number as a format of "#,###.##", rounding it to a certain number of decimal places
INSERT	Inserts a substring into a string at a specified position for a certain number of characters
LCASE	Converts a string to lower-case
LEFT	Extracts a substring from a string (starting from left)

LENGTH	Returns the length of the specified string (in bytes)
LOCATE	Returns the position of the first occurrence of a substring in a string
LOWER	Converts a string to lower-case
LPAD	Returns a string that is left-padded with a specified string to a certain length
LTRIM	Removes leading spaces from a string
MID	Extracts a substring from a string (starting at any position)
POSITION	Returns the position of the first occurrence of a substring in a string
REPEAT	Repeats a string a specified number of times
REPLACE	Replaces all occurrences of a specified string
REVERSE	Reverses a string and returns the result

RIGHT	Extracts a substring from a string (starting from right)
RPAD	Returns a string that is right-padded with a specified string to a certain length
RTRIM	Removes trailing spaces from a string
SPACE	Returns a string with a specified number of spaces
STRCMP	Tests whether two strings are the same
SUBSTR	Extracts a substring from a string (starting at any position)
SUBSTRING	Extracts a substring from a string (starting at any position)
TRIM	Removes leading and trailing spaces from a string
UPPER	Converts a string to upper-case

Regular expression metacharacters

What we looked at in the above example is the simplest form of a regular expression. Let's now look at more advanced regular expression pattern matches. Suppose we want to search for movie titles that start with the pattern "code" only using a regular expression, how would we go about it?

The answer is metacharacters. They allow us to fine tune our pattern search results using regular expressions.

--------To find all employees whose name starts with any character in range A to H

SELECT * FROM `movies` WHERE title REGEXP '^[A-H]';

Date functions in MySQL

MySQL String Functions

ASCII()

This function returns the numeric value of the leftmost character of the string str. Returns 0 if str is the empty string. Returns NULL if str is NULL
Syntax : ASCII(str)
Example : SELECT ASCII('2');
Output : 50
Example : SELECT ASCII(2);
Output : 50
Example : SELECT ASCII('An');
Output : 65

BIN()

Returns a string representation of the binary value of N, where N is a longlong (BIGINT) number. Returns NULL if N is NULL.
Syntax : BIN(N)
Example : SELECT BIN(12);
Output : 1100

BIT_LENGTH()

Returns the length of the string str in bits.
Syntax : BIT_LENGTH(str)
Example : SELECT BIT_LENGTH('text');
Output : 32

CHAR()

CHAR() interprets each argument N as an integer and returns a string consisting of the characters select code values of those integers. NULL values are skipped.
Syntax : CHAR(N,... [USING charset_name])
Example : SELECT CHAR(77,121,83,81,'76');
Output : MySQL
Example : SELECT CHAR(77,77.3,'77.3');
Output : MMM

CHAR_LENGTH()

Returns the length of the string str, measured in characters. A multi-byte character counts as a single character. This means that for a string containing five 2-byte characters, LENGTH() returns 10, whereas CHAR_LENGTH() returns 5.
Syntax : CHAR_LENGTH(str)
Example : SELECT CHAR_LENGTH('test string');
Output : 11

CONCAT()

Returns the string that results from concatenating one or more arguments. If all arguments are nonbinary strings, the result is a nonbinary string. If the arguments include any binary strings, the result is a binary string. A numeric argument is converted to its equivalent nonbinary string form.
Syntax : CONCAT(str1,str2,...)
Example : SELECT CONCAT('w3resource','.','com');
Output : w3resource.com

CONCAT_WS()

CONCAT_WS() stands for Concatenate With Separator and is a special form of CONCAT(). The first argument is the separator for the rest of the

arguments. The separator is added between the strings to be concatenated. The separator can be a string, as can the rest of the arguments. If the separator is NULL, the result is NULL.
Syntax : CONCAT_WS(separator,str1,str2,...)
Example : SELECT CONCAT_WS(',','1st string','2nd string');
Output : 1st string,2nd string

ELT()

ELT() returns the Nth element of the list of strings: str1 if N = 1, str2 if N = 2, and so on. Returns NULL if N is less than 1 or greater than the number of arguments. ELT() is the complement of FIELD().
Syntax : ELT(N,str1,str2,str3,...)
Example : SELECT ELT(4,'this','is','the','elt');
Output : elt

FIELD()

Returns the index (position) of str in the str1, str2, str3, ... list. Returns 0 if str is not found.
Syntax : FIELD(str,str1,str2,str3,...)
Example : SELECT FIELD('ank', 'b', 'ank', 'of', 'monk');
Output : 2

FIND_IN_SET()

Returns a value in the range of 1 to N if the string str is in the string list strlist consisting of N substrings. A string list is a string composed of substrings separated by “,” characters.
Returns NULL if either argument is NULL. This function does not work properly if the first argument contains a comma (“,”) character.
Syntax : FIND_IN_SET(str,strlist)
Example : SELECT FIND_IN_SET('ank','b,ank,of,monk');
Output : 2

FORMAT()

Formats the number X to a format like '#,###,###.##', rounded to D decimal places, and returns the result as a string. If D is 0, the result has no decimal point or fractional part.
Syntax : FORMAT(X,D)
Example : SELECT FORMAT(12332.123456, 4);
Output : 12,332.1235

Example : SELECT FORMAT(12332.1,4);
Output : 12,332.1000
Example : SELECT FORMAT(12332.2,0);
Output : 12,332

HEX()

MySQL HEX() returns a string representation of hexadecimal value of a decimal or string value specified as argument.
If the argument is a string, each character in the argument is converted to two hexadecimal digits.
If the argument is decimal, the function returns a hexadecimal string representation of the argument , and treated as a longlong(BIGINT) number.
Syntax : HEX(str), HEX(N)
Example : SELECT HEX(157);
Output : 9D

INSERT()

Returns the string str, with the substring beginning at position pos and len characters long replaced by the string newstr. Returns the original string if pos is not within the length of the string. Replaces the rest of the string from position pos if len is not within the length of the rest of the string.
Syntax : INSERT(str,pos,len,newstr)
Example : SELECT INSERT('Originalstring', 4, 5, ' insert ');
Output : Ori insert string
Example : SELECT INSERT('Originalstring', -3, 5, ' insert ');
Output : Originalstring

INSTR()

MySQL INSTR() takes a string and a substring of it as arguments, and returns an integer which indicates the position of the first occurrence of the substring within the string
Syntax : INSTR(str,substr)
Example : SELECT INSTR('myteststring','st');
Output : 5

LCASE()

MySQL LCASE() converts the characters of a string to lower case characters.
Syntax : LCASE(str)

Example : SELECT LCASE('MYTESTSTRING');
Output : myteststring

LEFT()

MySQL LEFT() returns a specified number of characters from the left of a given string. Both the number and the string are supplied in the arguments as str and len of the function.
Syntax : LEFT(str,len)
Example : SELECT LEFT('w3resource', 3);
Output : w3r

LENGTH()

MySQL LENGTH() returns the length of a given string.
Syntax : LENGTH(str)
Example : SELECT LENGTH('text');
Output : 4

LOCATE()

MySQL LOCATE() returns the position of the first occurrence of a string within a string. Both of these strings are passed as arguments. An optional argument may be used to specify from which position of the string (i.e. string to be searched) searching will start. If this position is not mentioned, searching starts from the beginning.
Syntax : LOCATE(substr,str,pos)
Example : SELECT LOCATE('st','myteststring');
Output : 5

LOWER()

MySQL LOWER() converts all the characters in a string to lowercase characters.
Syntax : LOWER(str)
Example : SELECT LOWER('MYTESTSTRING');
Output : myteststring

LPAD()

MySQL LPAD() left pads a string with another string. The actual string, a number indicating the length of the padding in characters (optional) and

the string to be used for left padding - all are passed as arguments.
Syntax : LPAD(str,len,padstr)
Example : SELECT LPAD('Hello',10,'*');
Output : *****Hello
Example : SELECT LPAD('hi',1,'**');
Output : h

LTRIM(str)

MySQL LTRIM() removes the leading space characters of a string passed as argument.
Syntax : LTRIM(str)
Example : SELECT LTRIM(' Hello')
Output : Hello (leading spaces have been exclude)

MID()

MySQL MID() extracts a substring from a string. The actual string, position to start extraction and length of the extracted string - all are specified as arguments.
Syntax : MID(str,pos,len)
Example : SELECT MID('w3resource',4,3);
Output : eso

OCT()

Returns a string representation of the octal value of N, where N is a longlong (BIGINT) number. Returns NULL if N is NULL.
Syntax : OCT(N)
Example : SELECT OCT(12);
Output : 14

ORD()

MySQL ORD() returns the code for the leftmost character if that character is a multi-byte (sequence of one or more bytes) one. If the leftmost character is not a multibyte character, ORD() returns the same value as the ASCII() function.
Syntax : ORD(str)
Example : SELECT ORD("w3resource");
Output : 119

POSITION()

MySQL POSITION() returns the position of a substring within a string..
Syntax : POSITION(substr IN str)
Example : SELECT POSITION("ou" IN "w3resource");
Output : 6

QUOTE()

Quotes a string to produce a result that can be used as a properly escaped data value in an SQL statement. The string is returned enclosed by single quotation marks and with each instance of backslash (“\”), single quote (“”), ASCII NUL, and Control+Z preceded by a backslash. If the argument is NULL, the return value is the word “NULL” without enclosing single quotation marks.
Syntax : QUOTE(str)
Example : SELECT QUOTE('w3re''source');
Output : 'w3re\'source'

REPEAT()

MySQL REPEAT() repeats a string for a specified number of times.
The function returns NULL either any either of the arguments are NULL.
Syntax : REPEAT(str,count)
Example : SELECT REPEAT('**-',5);
Output : **-**-**-**-**-

REPLACE()

MySQL REPLACE() replaces all the occurrences of a substring within a string.
Syntax : REPLACE(str,from_str,to_str)
Example : SELECT REPLACE('w3resource','ur','r');
Output : w3resorce

REVERSE()

Returns a given string with the order of the characters reversed.
Syntax : REVERSE(str)
Example : SELECT REVERSE('w3resource');
Output : ecruoser3w

RIGHT()

MySQL RIGHT() extracts a specified number of characters from the right side of a given string.
Syntax : RIGHT(str,len)
Example : SELECT RIGHT('w3resource',8);
Output : resource

RPAD()

MySQL RPAD() function pads strings from right. The actual string which is to be padded as str, length of the string returned after padding as len and string which is used for padding as padstr is used as a parameters within the argument.
Syntax : RPAD(str,len,padstr)
Example : SELECT RPAD('w3resource',15,'*');
Output : w3resource*****

RTRIM()

MySQL RTRIM() removes the trailing spaces from a given string.
Syntax : RTRIM(str)
Example : SELECT RTRIM('w3resource ');
(excludes the trailing spaces) Output : w3resource

SOUNDEX()

MySQL SOUNDEX() function returns soundex string of a string.
Soundex is a phonetic algorithm for indexing names after English pronunciation of sound. You can use SUBSTRING() on the result to get a standard soundex string. All non-alphabetic characters in str are ignored. All international alphabetic characters outside the A-Z range are treated as vowels.
Syntax : SOUNDEX(str)
Example : SELECT SOUNDEX('w3resource');
Output : w6262

SPACE()

MySQL SPACE() returns the string containing a number of spaces as specified in the argument.

Syntax : SPACE(N)
Example : SELECT 'start', SPACE(10), 'end';
Output : start SPACE(10) end

```
mysql> SELECT 'start', SPACE(10), 'end';
+-------+------------+-----+
| start | SPACE(10)  | end |
+-------+------------+-----+
| start |            | end |
+-------+------------+-----+
1 row in set (0.00 sec)
```

SUBSTR()

MySQL SUBSTR() returns the specified number of characters from a particular position of a given string. SUBSTR() is a synonym for SUBSTRING().
Syntax : SUBSTR(str,pos,len)
Example : SELECT SUBSTR('w3resource',4,3);
Output : eso

SUBSTRING()

MySQL SUBSTRING() returns a specified number of characters from a particular position of a given string.
Syntax : SUBSTRING(str,pos,len)
Example : SELECT SUBSTRING('w3resource',4,3);
Output : eso
Example : SELECT SUBSTRING('w3resource.com',5);
Output : source.com
Example : SELECT SUBSTRING('w3resource.com',-5);
Output : e.com

SUBSTRING_INDEX()

MySQL SUBSTRING_INDEX() returns the substring from the given string before a specified number of occurrences of a delimiter.
Returns from the left of the final delimiter if the number is positive and right of the final delimiter when the number is negative.
If the number is greater than the number of occurrence of delimiter, the

returned substring will be the total string. If the specified number is 0, nothing will be fetched from the given string.
Syntax : SUBSTRING_INDEX(str,delim,count)
Example : SELECT SUBSTRING_INDEX('www.mytestpage.info','.',2);
Output : www.mytestpage

TRIM()

MySQL TRIM() function returns a string after removing all prefixes or suffixes from the given string.
Syntax : TRIM([{BOTH | LEADING | TRAILING} [remstr] FROM] str)
Example : SELECT TRIM(' trim ');
Output : trim (leading and trailing space removed)
Example : SELECT TRIM(LEADING 'leading' FROM 'leadingtext');
Output : text
Example : SELECT TRIM(BOTH 'leadtrail' FROM 'leadtrailtextleadtrail');
Output : text

UNHEX()

MySQL UNHEX() function performs the opposite operation of HEX(). This function interprets each pair of hexadecimal digits (in the argument) as a number and converts it to a character.
Syntax : UNHEX(str)
Example : SELECT UNHEX('4D7953514C');
Output : MySQL
Example : SELECT UNHEX(HEX('MySQL'));
Output : MySQL

UPPER()

MySQL UPPER() converts all the characters in a string to uppercase characters.
Syntax : UPPER(str)
Example : SELECT UPPER('myteststring');
Output : MYTESTSTRING

To find nth Highest record

Note : OFFSET starts from 0th position, and hence use N-1 rule here

---------to find nth Salary: (offset will skip the rows counting starts with 0 hence use n-1)

SELECT DISTINCT Salary FROM emp ORDER BY Salary DESC LIMIT 1 OFFSET (7)

---------- to find 8th highest salary:

SELECT DISTINCT Salary FROM emp ORDER BY Salary DESC LIMIT 1 OFFSET 7

----------to find first 10 rows

Select *

from emp

limit 10;

---------to find rows starting from 3 rd onward 7 rows (counting starts with 0)

Select *

from emp

limit 3,7

Data types in mysql

- Numeric
- Date and Time
- String Types.

Let us now discuss them in detail.

Numeric Data Types

MySQL uses all the standard ANSI SQL numeric data types, so if you're coming to MySQL from a different database system, these definitions will look familiar to you. The following list shows the common numeric data types and their descriptions −

- **INT** − A normal-sized integer that can be signed or unsigned. If signed, the allowable range is from -2147483648 to 2147483647. If unsigned, the allowable range is from 0 to 4294967295. You can specify a width of up to 11 digits.
- **TINYINT** − A very small integer that can be signed or unsigned. If signed, the allowable range is from -128 to 127. If unsigned, the allowable range is from 0 to 255. You can specify a width of up to 4 digits.

- **SMALLINT** − A small integer that can be signed or unsigned. If signed, the allowable range is from -32768 to 32767. If unsigned, the allowable range is from 0 to 65535. You can specify a width of up to 5 digits.
- **MEDIUMINT** − A medium-sized integer that can be signed or unsigned. If signed, the allowable range is from -8388608 to 8388607. If unsigned, the allowable range is from 0 to 16777215. You can specify a width of up to 9 digits.
- **BIGINT** − A large integer that can be signed or unsigned. If signed, the allowable range is from -9223372036854775808 to 9223372036854775807. If unsigned, the allowable range is from 0 to 18446744073709551615. You can specify a width of up to 20 digits.
- **FLOAT(M,D)** − A floating-point number that cannot be unsigned. You can define the display length (M) and the number of decimals (D). This is not required and will default to 10,2, where 2 is the number of decimals and 10 is the total number of digits (including decimals). Decimal precision can go to 24 places for a FLOAT.
- **DOUBLE(M,D)** − A double precision floating-point number that cannot be unsigned. You can define the display length (M) and the number of decimals (D). This is not required and will default to 16,4, where 4 is the number of decimals. Decimal precision can go to 53 places for a DOUBLE. REAL is a synonym for DOUBLE.
- **DECIMAL(M,D)** − An unpacked floating-point number that cannot be unsigned. In the unpacked decimals, each decimal corresponds to one byte. Defining the display length (M) **Date and Time Types**

The MySQL date and time datatypes are as follows −

- **DATE** − A date in YYYY-MM-DD format, between 1000-01-01 and 9999-12-31. For example, December 30th, 1973 would be stored as 1973-12-30.
- **DATETIME** − A date and time combination in YYYY-MM-DD HH:MM:SS format, between 1000-01-01 00:00:00 and 9999-12-31 23:59:59. For example, 3:30 in the afternoon on December 30th, 1973 would be stored as 1973-12-30 15:30:00.
- **TIMESTAMP** − A timestamp between midnight, January 1st, 1970 and sometime in 2037. This looks like the previous DATETIME format, only without the hyphens between numbers; 3:30 in the afternoon on December 30th, 1973 would be stored as 19731230153000 (YYYYMMDDHHMMSS).
- **TIME** − Stores the time in a HH:MM:SS format.
- **YEAR(M)** − Stores a year in a 2-digit or a 4-digit format. If the length is specified as 2 (for example YEAR(2)), YEAR can be between 1970 to 2069 (70 to 69). If the length is specified as 4, then YEAR can be 1901 to 2155. The default length is 4.

String Types

Although the numeric and date types are fun, most data you'll store will be in a string format. This list describes the common string datatypes in MySQL.

- **CHAR(M)** − A fixed-length string between 1 and 255 characters in length (for example CHAR(5)), right-padded with spaces to the specified length when stored. Defining a length is not required, but the default is 1.

- **VARCHAR(M)** − A variable-length string between 1 and 255 characters in length. For example, VARCHAR(25). You must define a length when creating a VARCHAR field.

- **BLOB or TEXT** − A field with a maximum length of 65535 characters. BLOBs are "Binary Large Objects" and are used to store large amounts of binary data, such as images or other types of files. Fields defined as TEXT also hold large amounts of data. The difference between the two is that the sorts and comparisons on the stored data are **case sensitive** on BLOBs and are **not case sensitive** in TEXT fields. You do not specify a length with BLOB or TEXT.
- **TINYBLOB or TINYTEXT** − A BLOB or TEXT column with a maximum length of 255 characters. You do not specify a length with TINYBLOB or TINYTEXT.
- **MEDIUMBLOB or MEDIUMTEXT** − A BLOB or TEXT column with a maximum length of 16777215 characters. You do not specify a length with MEDIUMBLOB or MEDIUMTEXT.
- **LONGBLOB or LONGTEXT** − A BLOB or TEXT column with a maximum length of 4294967295 characters. You do not specify a length with LONGBLOB or LONGTEXT.
- **ENUM** − An enumeration, which is a fancy term for list. When defining an ENUM, you are creating a list of items from which the value must be selected (or it can be NULL). For example, if you wanted your field to contain "A" or "B" or "C", you would define your ENUM as ENUM ('A', 'B', 'C') and only those values (or NULL) could ever populate that field.

To create table

----

1. CREATE TABLE IF NOT EXISTS newpublisher

```
(pub_id varchar(8) NOT NULL UNIQUE DEFAULT '' ,
pub_name varchar(50) NOT NULL  DEFAULT '' ,
pub_city varchar(25) NOT NULL  DEFAULT '' ,
country varchar(25) NOT NULL DEFAULT 'India',
```

```
country_office varchar(25) ,
no_of_branch int(3),
pub_date date
CHECK ((country='India' AND pub_city='Mumbai')
OR (country='India' AND pub_city='New Delhi')) ,
PRIMARY KEY (pub_id));
```

2. CREATE TABLE IF NOT EXISTS newauthor

(id int NOT NULL AUTO_INCREMENT,

aut_id varchar(8),

aut_name varchar(50),

country varchar(25),

home_city varchar(25) NOT NULL,

PRIMARY KEY (id));

Using Forign key in mysql

CONSTRAINT constraint_name

FOREIGN KEY foreign_key_name (columns)

REFERENCES parent_table(columns)

ON DELETE action

ON UPDATE action

Action	
	The ON DELETE clause allows you to define what happens to the records in the child table when the records in the parent table are deleted. If you omit the ON DELETE clause and delete a record in the parent table that has records in

	the child table refer to, MySQL will reject the deletion.
ON DELETE CASCADE	delete records in the child table that refers to a record in the parent table when the record in the parent table is deleted
ON DELETE SET NULL	set the foreign key column values in the child table to NULL
ON DELETE NO ACTION or ON DELETE RESTRICT	will reject the deletion.

15 hr Mumbai

100 Rajan lkdjfkjs NULL

Let’s examine the syntax in greater detail:

The CONSTRAINT clause allows you to define constraint name for the foreign key constraint. If you omit it, MySQL will generate a name automatically.

The FOREIGN KEY clause specifies the columns in the child table that refers to primary key columns in the parent table. You can put a foreign key name after

FOREIGN KEY clause or leave it to let MySQL create a name for you. Notice that MySQL automatically creates an index with the foreign_key_name name.

The REFERENCES clause specifies the parent table and its columns to which the columns in the child table refer. The number of columns in the child table and parent table specified in the FOREIGN KEY and REFERENCES must be the same.

Examples:

```
CREATE DATABASE IF NOT EXISTS dbdemo;

USE dbdemo;

CREATE TABLE categories(
   cat_id int not null auto_increment primary key,
   cat_name varchar(255) not null,
   cat_description text
);

CREATE TABLE products(
   prd_id int not null auto_increment primary key,
   prd_name varchar(355) not null,
   prd_price decimal,
   cat_id int not null,
   FOREIGN KEY fk_cat(cat_id)
```

```
  REFERENCES categories(cat_id)
  ON UPDATE CASCADE
  ON DELETE RESTRICT
)ENGINE=InnoDB;
```

```
Create table mytable(
Id int primary key auto_increment,
Name varchar(10),
Addr varchar(20));
----- to assign starting value
ALTER TABLE mytable AUTO_INCREMENT=1001
----- to set step for increament

Insert into mytable values(default,'Kishori','Aundh')
Insert into mytable(name,addr) values('Kishori','Aundh')
```

To delete unique constraint ----delete index will delete the constraint

```
create table test_mytab(unid int,activity_name varchar(100),CONSTRAINT
activty_uqniue UNIQUE(activity_name),primary key (unid));

alter table unique_constraints
drop index activty_uqniue;        ----------delete unique constraint
```

To delete constraint

ALTER TABLE `table_name` DROP FOREIGN KEY `id_name_fk`;

ALTER TABLE `table_name` DROP INDEX `id_name_fk`;

To add constraint in existing table

Lets create properties and user table and then add constraint in user table

```
CREATE TABLE Properties
(
    ID int AUTO_INCREMENT,
    language int,
    stonecolor int,
    gamefield int,
    UserID int,
    PRIMARY KEY(ID),
    FOREIGN KEY (language) REFERENCES Language(ID),
    FOREIGN KEY(stonecolor) REFERENCES StoneColor(ID),
    FOREIGN KEY(gamefield) REFERENCES GameField(ID)
) ENGINE = INNODB;


CREATE TABLE User
(
    ID int AUTO_INCREMENT,
    vorname varchar(30) NOT NULL,
    name varchar(30) NOT NULL,
    email varchar(40) NOT NULL,
```

```
    password varchar(40) NOT NULL,
    nickname varchar(15) NOT NULL,
    score int,
    isadmin int DEFAULT 0,
    gamesPlayed int,
    properties int NOT NULL,
    PRIMARY KEY(ID),
    UNIQUE (email),
    UNIQUE (nickname)

) ENGINE = INNODB;
```

To alter table

---------- To add constraint

```
ALTER TABLE User
    ADD CONSTRAINT userProperties
    FOREIGN KEY(properties)
    REFERENCES Properties(ID)
```

---------- To add a column

Add column in table

Syntax

------The syntax to add a column in a table in MySQL (using the ALTER TABLE statement) is:

ALTER TABLE table_name

ADD new_column_name column_definition

[FIRST | AFTER column_name];

Add multiple columns in table

Syntax

---------The syntax to add multiple columns in a table in MySQL (using the ALTER TABLE statement) is:

ALTER TABLE table_name

ADD new_column_name column_definition

[FIRST | AFTER column_name],

ADD new_column_name column_definition

[FIRST | AFTER column_name],

...

;

Example

-------To add lastname column after contacted

ALTER TABLE contacts

ADD lastname varchar(40) NOT NULL

AFTER contactid;

-------To add multiple columns

ALTER TABLE contacts

ADD lastname varchar(40) NOT NULL

FIRST,

ADD first_name varchar(35) NULL

AFTER last_name;

----------Add Partitions in table

```
ALTER TABLE orders PARTITION BY LIST(st) (
        PARTITION p0 VALUES IN (20,10),
        PARTITION p1 VALUES IN (0,-10)
);
```

The ADD is extraneous - the syntax is essentially identical to the CREATE TABLE statement.

Make sure you have a good, restorable backup before doing this.

Log:

```
mysql> create table orders (id int, st int, whatever varchar(10), primary key (id));
Query OK, 0 rows affected (0.06 sec)

mysql> ALTER TABLE orders DROP PRIMARY KEY, ADD PRIMARY KEY(id, st);
Query OK, 0 rows affected (0.06 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> ALTER TABLE orders PARTITION BY LIST(st) (
    ->          PARTITION p0 VALUES IN (20,10),
    ->          PARTITION p1 VALUES IN (0,-10)
    -> );
Query OK, 0 rows affected (0.06 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

---------- To modify a column

Modify column in table

Syntax

The syntax to modify a column in a table in MySQL (using the ALTER TABLE statement) is:

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

Example:

```
ALTER TABLE contacts
  MODIFY last_name varchar(50) NULL;
```

-------To drop column

```
ALTER TABLE table_name
  DROP COLUMN column_name;
```

Example:

```
ALTER TABLE contacts
  DROP COLUMN contact_type;
```

---------- To rename column

```
ALTER TABLE table_name
  CHANGE COLUMN old_name new_name
    column_definition
    [ FIRST | AFTER column_name ]
```

Example: To rename the column contact_type to ctype.

```
ALTER TABLE contacts
  CHANGE COLUMN contact_type ctype
    varchar(20) NOT NULL;
```

-------To Rename table

Syntax

The syntax to rename a table in MySQL is:

```
ALTER TABLE table_name
  RENAME TO new_table_name;
```

Example:

```
ALTER TABLE contacts
  RENAME TO people;
```

Create index

```
CREATE  UNIQUE INDEX newautid ON newauthor(aut_id);
```

Create index salidx on emp(sal,ename);

case statement in MYSQL

Syntax:

CASE value WHEN [compare_value] THEN result

[WHEN [compare_value] THEN result ...]

[ELSE result]

END

OR

CASE WHEN [condition] THEN result

[WHEN [condition]

THEN result ...]

[ELSE result]

END

- The first syntax returns the result where value=compare_value.
- The second syntax returns the result for the first condition that is true.
- The list of corresponding SQL statements will execute when a search condition evaluates to true.
- The statement list in ELSE part will execute when no search condition matches.
- If there is no matching value found in the ELSE part, NULL will be returned.
- Each statement list can contain one or more statements and no empty statement list is allowed.

1. SELECT OrderID, Quantity,

   CASE

```
    WHEN Quantity > 30 THEN "The quantity is greater than 30"
    WHEN Quantity = 30 THEN "The quantity is 30"
    ELSE "The quantity is something else"
END
FROM OrderDetails;
```

2.
```
SELECT CustomerName, City, Country
FROM Customers
ORDER BY
(CASE
    WHEN City IS NULL THEN Country
    ELSE City
END);
```

3. Suppose you want to sort the customers by state, and if the state is NULL, you want to use the country as the sorting criterion instead. To achie SELECT Name, RatingID AS Rating,

```
-> CASE RatingID
->   WHEN 'R' THEN 'Under 17 requires an adult.'
->   WHEN 'X' THEN 'No one 17 and under.'
->   WHEN 'NR' THEN 'Use discretion when renting.'
->   ELSE 'OK to rent to minors.'
-> END AS Policy
-> FROM DVDs
-> ORDER BY Name
```

----------you can use the first form of the CASE expression as follows:

```
SELECT
  customerName, state, country
FROM
```

customers

ORDER BY (CASE

WHEN state IS NULL THEN country

ELSE state

END);

Select cid,cname,nop,case when nop<10 then “few participants cancel the course”

When nop < 20 then “moderate participants wait for 10 mins”

When nop < 30 then “normal count wait for 5 mins”

Else “start session” end ‘status’

From course;

Create view myview

As

Select ename,deptno

From emp

Where deptno=10

With check constraint cn;

------with read only constraint cn;

Insert into myview values)

-----------To see the base query for views

If we want to see the SQL statements that make up a particular view, we can use the script shown below to do that.

SHOW CREATE VIEW `accounts_v_members`;

Trigger

Create trigger mytriger on emp

Instead of insert

As

Insert into emp values(default,NEW.ename

End tigger;

Select * From EmpIndia

union

Select * From EmpUK

Union

Select * From EmpUS;

Select * From EmpIndia

intersect

Select * From EmpUK

Select * From EmpIndia

minus

Select * From EmpUK

select *

-> from emp

-> where deptno=10

```
-> union
-> select *
-> from emp
-> where deptno=20;
```

PL SQL

Writing procedures in mysql

Create table query

```
/*Create Employee database for demo */
    CREATE DATABASE Employee;
    /*Create sample E
    mployeeDetails table.*/
    CREATE TABLE emp
    (
    EmpID INTEGER
    ,EmpName VARCHAR(50)
    ,EmailAddress VARCHAR(50)
    ,CONSTRAINT pk_tbl_EmployeeDetails_EmpID PRIMARY KEY (EmpID)
    );
```

----------Procedure examples

Parameters --- 3 types -----in -----read only , out ---- write only , inout ---read and write

By default parameters are in parameters

```
delimiter //
mysql> CREATE PROCEDURE simpleproc (OUT param1 INT)
    -> BEGIN
    ->   SELECT COUNT(*) INTO param1 FROM emp;
    -> END//

mysql> delimiter ;
```

mysql> CALL simpleproc(@a);

To see value of @a

Select @a;

--------Mysql If else statement

```
IF expression THEN
  statements;
ELSE
  else-statements;
END IF;
```

Using If ----else--------

IF expression THEN

  statements;

ELSEIF elseif-expression THEN

  elseif-statements;

...

ELSE

  else-statements;

END IF;

Example-------

```
DELIMITER $$

CREATE PROCEDURE GetCustomerLevel(
   in  p_customerNumber int(11),
   out p_customerLevel  varchar(10))
BEGIN
   DECLARE vcreditlim double;

   SELECT creditlimit INTO vcreditlim
   FROM customers
   WHERE customerNumber = p_customerNumber;

   IF creditlim > 50000 THEN
         SET p_customerLevel = 'PLATINUM';
   ELSEIF (creditlim <= 50000 AND creditlim >= 10000) THEN
        SET p_customerLevel = 'GOLD';
```

```
  ELSEIF creditlim < 10000 THEN
      SET p_customerLevel = 'SILVER';
  END IF;
```

END$$

While loop----------

```
WHILE expression DO
  statements
END WHILE
```

To display number 1,2,3,4,5

```
DELIMITER $$
DROP PROCEDURE IF EXISTS test_mysql_while_loop$$
CREATE PROCEDURE test_mysql_while_loop()
BEGIN
DECLARE x  INT;
DECLARE str  VARCHAR(255);
SET x = 1;
SET str =  '';
WHILE x  <= 5 DO
SET  str = CONCAT(str,x,',');
SET  x = x + 1;
END WHILE;
SELECT str;
END$$
DELIMITER ;
```

REPEAT loop

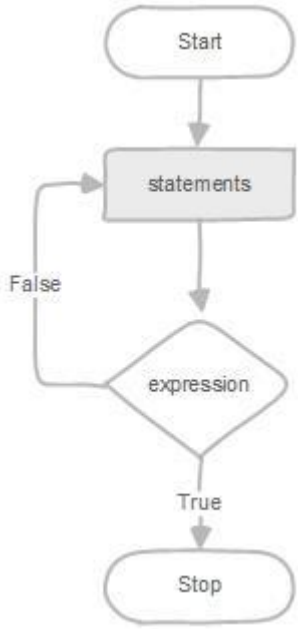
Syntax :-

```
    REPEAT
            statements;
    UNTIL expression
END REPEAT
```

First, MySQL executes the statements, and then it evaluates the expression. If the expression evaluates to FALSE, MySQL executes the statements repeatedly until the expression evaluates to TRUE.

Because the REPEAT loop statement checks the expression after the execution of statempts therefore the REPEAT loop statement is also known as the post-test loop.

The following flowchart illustrates the REPEAT loop statement:



using the repeate loop statement:

```
DELIMITER $$
DROP PROCEDURE IF EXISTS mysql_test_repeat_loop$$
CREATE PROCEDURE mysql_test_repeat_loop()
BEGIN
DECLARE x INT;
DECLARE str VARCHAR(255);

SET x = 1;
     SET str =  '';

REPEAT
    SET  str = CONCAT(str,x,',');
    SET  x = x + 1;
 UNTIL x  > 5
 END REPEAT;

 SELECT str;
END$$
DELIMITER ;
```

It is noted that there is no semicolon (;) in the `UNTIL` expression.

--------to call mysql procedure

```
CALL mysql_test_repeat_loop();
```

LOOP, LEAVE and ITERATE statements

There are two statements that allow you to control the loop:

- The **leave** statement allows you to exit the loop immediately without waiting for checking the condition.
- The ITERATE statement allows you to skip the entire code under it and start a new iteration. similar to the `continue` statement C/C++, Java, etc.

MySQL also gives you a loop statement that executes a block of code repeatedly with an additional flexibility of using a loop label.
The following is an example of using the loop statement.

```
CREATE PROCEDURE test_mysql_loop()
BEGIN
DECLARE x  INT;
DECLARE str  VARCHAR(255);
  SET x = 1;
SET str =  '';
loop_label:  LOOP
IF  x > 10 THEN
LEAVE  loop_label;
END  IF;
SET  x = x + 1;
IF  (x mod 2=0) THEN
        ITERATE  loop_label;
ELSE
          SET  str = CONCAT(str,x,',');
END  IF;
END LOOP;
SELECT str;
END;
```

- The stored procedure only constructs a string with even numbers e.g., 2, 4, 6, etc. We put a loop_label loop label before the LOOPstatement.

- If the value of x is greater than 10, the loop is terminated because of the LEAVE statement.
- If the value of the x is an odd number, the ITERATE statement ignores everything below it and starts a new iteration.
- If the value of the x is an even number, the block in the ELSE statement will build the string with even numbers.

Declaring variables

To declare a variable inside a stored procedure, you use the DECLARE statement as follows:

```
1    DECLARE variable_name datatype(size) DEFAULT default_value;
```

Let's examine the statement above in more detail:

```
1    DECLARE total_sale INT DEFAULT 0;
```

MySQL allows you to declare two or more variables that share the same data type using a singl e DECLARE statement as following:

```
DECLARE x, y INT DEFAULT 0;
```

We declared two integer variables x and y, and set their default values to zero.

Assigning variables

Once you declared a variable, you can start using it. To assign a variable another value, you use the SET statement, for example:

```
DECLARE total_count INT DEFAULT 0;
SET total_count = 10;
```

The value of the total_count variable is 10 after the assignment.
Besides the SET statement, you can use the SELECT INTO statement to assign the result of a q uery, which returns a scalar value, to a variable. Ssee the following example:

```
DECLARE total_products INT DEFAULT 0

SELECT COUNT(*) INTO total_products
FROM products
```

In the example above:

- First, we declare a variable named total_products and initialize its value to 0.
- Then, we used the SELECT INTO statement to assign the total_products variable the n umber of products that we selected from the products table .

Variables scope

A variable has its own scope that defines its lifetime. If you declare a variable inside a stored pr ocedure, it will be out of scope when the END statement of stored procedure reached.
If you declare a variable inside BEGIN END block, it will be out of scope if the END is reached. You can declare two or more variables with the same name in different scopes because a varia ble is only effective in its own scope. However, declaring variables with the same name in diffe rent scopes is not good programming practice.
A variable that begins with the @ sign is session variable. It is available and accessible until the session ends.

Introduction to MySQL cursor

To handle a result set inside a stored procedure, you use a cursor.
A cursor allows you to iterate a set of rows returned by a query and process each row accordin gly.
MySQL cursor is read-only, non-scrollable and asensitive.

- Read only: you cannot update data in the underlying table through the cursor.
- Non-scrollable: you can only fetch rows in the order determined by the SELECT statem ent. You cannot fetch rows in the reversed order. In addition, you cannot skip rows or jump to a specific row in the result set.
- A sensitive: there are two kinds of cursors:
    - asensitive cursor and insensitive cursor. An asensitive cursor points to the act ual data, whereas an insensitive cursor uses a temporary copy of the data. An a sensitive cursor performs faster than an insensitive cursor because it does not have to make a temporary copy of data. However, any change that made to t he data from other connections will affect the data that is being used by an ase nsitive cursor, therefore, it is safer if you don't update the data that is being us ed by an asensitive cursor. MySQL cursor is asensitive.

You can use MySQL cursors in stored procedures, stored functions, and triggers.
Working with MySQL cursor
Step 1:
declare a cursor by using the DECLARE statement:

```
1    DECLARE cursor_name CURSOR FOR SELECT_statement;
```

The cursor declaration must be after any variable declaration. If you declare a cursor before va riables declaration, MySQL will issue an error. A cursor must always be associated with a SELE CT statement.
Step 2:
open the cursor by using the OPEN statement. The OPEN statement initializes the result set for the cursor, therefore, you must call the OPEN statement before fetching rows from the result s et.

```
1    OPEN cursor_name;
```

Then, you use the FETCH statement to retrieve the next row pointed by the cursor and move t he cursor to the next row in the result set.

```
1    FETCH cursor_name INTO variables list;
```

After that, you can check to see if there is any row available before fetching it.

Step 3:

Finally, you call the CLOSE statement to deactivate the cursor and release the memory associat ed with it as follows:

```
1	CLOSE cursor_name;
```

When the cursor is no longer used, you should close it.
When working with MySQL cursor, you must also declare a NOT FOUND handler to handle the situation when the cursor could not find any row. Because each time you call the FETCH state ment, the cursor attempts to read the next row in the result set. When the cursor reaches the e nd of the result set, it will not be able to get the data, and a condition is raised. The handler is u sed to handle this condition.
To declare a NOT FOUND handler, you use the following syntax:

```
1	DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_finished = 1;
```

Where finished is a variable to indicate that the cursor has reached the end of the result set.
Notice that the handler declaration must appear after variable and cursor declaration inside t he stored procedures.

MySQL Cursor Example
We are going to develop a stored procedure that builds an email list of all employees in the em ployee table in the MySQL sample database.
First, we declare some variables, a cursor for looping over the emails of employees, and a NOT FOUND handler:

```
DECLARE v_finished INTEGER DEFAULT 0;
DECLARE ename_list varchar(255) DEFAULT "";

-- declare cursor for employee email
DEClARE emp_cursor CURSOR FOR
 SELECT ename FROM employees;

-- declare NOT FOUND handler
DECLARE CONTINUE HANDLER
FOR NOT FOUND SET finished = 1;
```

Next, we open the emp_cursor by using the OPEN statement:

```
1	OPEN emp_cursor;
```

Then, we iterate the emp list, and concatenate all enames where each ename is separated by a semicolon(;):

```
 get_ename: LOOP
 FETCH emp_cursor INTO v_ename;
 IF finished = 1 THEN
 LEAVE get_ename;
 END IF;
 -- build ename list
 SET ename_list = CONCAT(v_ename,";",ename_list);
END LOOP get_email;
```

After that, inside the loop we used the v_finished variable to check if there is any ename in the l ist to terminate the loop.
Finally, we close the cursor using the CLOSE statement:

```
1	CLOSE ename_cursor;
```

The build_ename_list stored procedure is as follows:

```
DELIMITER $$

CREATE PROCEDURE build_ename_list (INOUT ename_list varchar(4000))
BEGIN

 DECLARE v_finished INTEGER DEFAULT 0;
 DECLARE v_ename varchar(100) DEFAULT "";

 -- declare cursor for employee email
 DEClARE ename_cursor CURSOR FOR
 SELECT ename FROM employees;

 -- declare NOT FOUND handler
 DECLARE CONTINUE HANDLER
      FOR NOT FOUND SET v_finished = 1;

 OPEN ename_cursor;

 get_ename: LOOP

 FETCH ename_cursor INTO v_ename;

 IF v_finished = 1 THEN
 LEAVE get_email;
 END IF;

 -- build email list
 SET ename_list = CONCAT(v_ename,";",ename_list);

 END LOOP get_ename;

 CLOSE ename_cursor;

END$$

DELIMITER ;
```

You can test the build_ename_list stored procedure using the following script:

```
SET @ename_list = "";
CALL build_ename_list(@ename_list);
SELECT @ename_list;
```

----------Example-2

```
DELIMITER //
 DROP PROCEDURE IF EXISTS sp_set_name //
 CREATE PROCEDURE sp_set_name ()
 BEGIN
 DECLARE done INT DEFAULT 0;
 DECLARE client_id INT;
 DECLARE cur1 CURSOR FOR SELECT id from clients WHERE name IS NULL;
 DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

OPEN cur1;
```

```
read_loop: LOOP
 IF done THEN
 LEAVE read_loop;
 END IF;
 FETCH cur1 INTO client_id;
 UPDATE clients SET name = (SELECT name from orders WHERE id = client_id)
 WHERE id = client_id;
 END LOOP;

close cur1;
 END;
 //
 DELIMITER ;
```

Functions in mysql--------

```
DELIMITER $$
CREATE FUNCTION calcProfit(cost FLOAT, price FLOAT) RETURNS DECIMAL(9,2)
BEGIN
  DECLARE profit DECIMAL(9,2);
  SET profit = price-cost;
  RETURN profit;
END$$
DELIMITER ;
```

-----call function using select statement

```
SELECT  *, calcProfit(prod_cost,prod_price) AS profit FROM products;
```

--------example to list offices from given country

```
DELIMITER //
CREATE PROCEDURE GetOfficeByCountry(IN countryName VARCHAR(255))
 BEGIN
 SELECT *
 FROM offices
 WHERE country = countryName;
 END //

DELIMITER ;
```

The countryName is the IN parameter of the stored procedure. Inside the stored procedure, w e select all offices that locate in the country specified by the countryName parameter.
Suppose, we want to get all offices in the USA, we just need to pass a value (USA) to the stored procedure as follows:

```
1	CALL GetOfficeByCountry('USA');

------------------example to find count by order status

DELIMITER $$
CREATE PROCEDURE CountOrderByStatus(IN orderStatus VARCHAR(25),OUT total INT)
BEGIN
SELECT count(orderNumber) INTO total
FROM orders
WHERE status = orderStatus;
END$$

DELIMITER ;


----------To call the procedure
CALL CountOrderByStatus('Shipped',@total);
SELECT @total;
CALL CountOrderByStatus('in process',@total);
SELECT @total AS  total_in_process;

DELIMITER $$
CREATE PROCEDURE set_counter(INOUT count INT(4),IN inc INT(4))
BEGIN
SET count = count + inc;
END$$
DELIMITER ;


To call procedure ------
SET @counter = 1;
CALL set_counter(@counter,1); -- 2
CALL set_counter(@counter,1); -- 3
CALL set_counter(@counter,5); -- 8
SELECT @counter; -- 8

Example --------
DELIMITER @@

DROP PROCEDURE IF EXISTS import_members@@
CREATE PROCEDURE import_members ()
BEGIN
  -- Declare loop constructs --
  DECLARE done INT DEFAULT FALSE;

  -- Declare Person variables --
  DECLARE my_person_id INT;
  DECLARE my_era_username VARCHAR(100);
  DECLARE my_last_name VARCHAR(50);
  DECLARE my_first_name VARCHAR(50);
  DECLARE my_email VARCHAR(100);
  DECLARE my_email_primary VARCHAR(50);
  DECLARE my_degree_id_1 INT;
  DECLARE my_degree_id_2 INT;
  DECLARE my_member_status INT;
```

```
  DECLARE my_user_id INT;
  DECLARE my_user_email VARCHAR(100);

  -- Declare Cursor --
  DECLARE member_cursor CURSOR FOR
    SELECT person_id, era_username, last_name, first_name,
        TRIM(email), TRIM(email_primary), degree_id_1, degree_id_2,
        member_status FROM z_data_person
    WHERE member_status IS NOT NULL;

  -- Declare Continue Handler --
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

  OPEN member_cursor;

  read_loop: LOOP

    -- Fetch data from cursor --
    FETCH member_cursor
    INTO my_person_id, my_era_username, my_last_name, my_first_name,
        my_email, my_email_primary, my_degree_id_1, my_degree_id_2,
        my_member_status;

    -- Exit loop if finished --
    IF done THEN
      LEAVE read_loop;
    END IF;

    -- Create User --
    SET my_user_id = SELECT (MAX(uid) + 1) FROM users;
    IF my_email_primary IS NOT NULL AND my_email_primary NOT LIKE '%null%' THEN
      SET my_user_email = my_email_primary;
    ELSE
      SET my_user_email = my_email;
    END IF;

    INSERT INTO `users` (`uid`, `name`, `pass`, `mail`, `created`)
    VALUES (my_user_id, my_era_username, SHA1(RAND()), my_user_email, UNIX_TIMESTAM
P());

    -- Create Member --
    INSERT INTO `members` (`uid`, `first_name`, `last_name`, `phone`,
        `member_category_id`, `subscription_weekly_email`,
        `subscription_monthly_newsletter`)
    VALUES (my_user_id, my_first_name, my_last_name, my_phone_num,
        my_member_status, my_weekly_com, my_monthly_com);

    -- Add Degrees --
    IF degree_id_1 IS NOT NULL THEN
      INSERT INTO member_degrees_held
      VALUES (my_user_id, my_degree_id_1);
    END IF;

    IF degree_id_2 IS NOT NULL THEN
      INSERT INTO member_degrees_held
```

```
        VALUES (my_user_id, my_degree_id_2);
    END IF;

    -- Add Areas of Expertise --
    INSERT INTO `member_expertises_held` (`uid`, `specialty_id`)
    SELECT (SELECT MAX(uid) FROM users), z.specialty_id
    FROM z_map_person_specialty AS z
    WHERE z.person_id = my_person_id;

  END LOOP read_loop;

  CLOSE member_cursor;

END; @@

DELIMITER ;
CALL import_members();
```

Triggers in mysql

What are MySQL triggers and how to use them?

The MySQL trigger is a database object that is associated with a table. It will be activated when a defined action is executed for the table. The trigger can be executed when you run one of the following MySQL statements on the table: INSERT, UPDATE and DELETE and it can be invoked before or after the event.

The main requirement for running such MySQL Triggers is having MySQL SUPERUSER privileg es.

Here is an example of a MySQL trigger:

- First we will create the table for which the trigger will be set

```
1    mysql> CREATE TABLE people (age INT, name varchar(150));
```

- Next we will define the trigger. It will be executed before every INSERT statement for t he people table:

```
        mysql> delimiter //
mysql> CREATE TRIGGER agecheck BEFORE INSERT ON people
       FOR EACH ROW
       IF NEW.age < 0 THEN
          SET NEW.age = 0;
      END IF;//
Query OK, 0 rows affected (0.00 sec)
mysql> delimiter ;
```

- We will insert two records to check the trigger functionality.

```
        mysql> INSERT INTO people VALUES (-20, 'Sid'), (30, 'Josh');

Query OK, 2 rows affected (0.00 sec)
Records: 2 Duplicates: 0 Warnings: 0
```

- At the end we will check the result.

mysql> SELECT * FROM people;
+———-+———-+
| age | name |
+———-+———-+
| 0 | Sid |
| 30 | Josh |
+———-+———-+
2 rows in set (0.00 sec)

To Remove trigger-------

DROP TRIGGER table_name.trigger_name;

To Display list of triggers-------

Show trigger

To see the definition of named trigger
SHOW CREATE TRIGGER trigger_name example

Syntax

CREATE TRIGGER trigger_name trigger_time trigger_event
ON table_name
FOR EACH ROW
BEGIN
...
END;

MySQL trigger example
Let's start creating a trigger in MySQL to log the changes of the employees table.

First, create a new table named employees_audit to keep the changes of the employee table. Th e following statement creates the employee_audit table.

```
CREATE TABLE employees_audit (
 id INT AUTO_INCREMENT PRIMARY KEY,
 employeeNumber INT NOT NULL,
 lastname VARCHAR(50) NOT NULL,
 changedat DATETIME DEFAULT NULL,
 action VARCHAR(50) DEFAULT NULL
);
```

create a BEFORE UPDATE trigger that is invoked before a change is made to the employees ta ble.

```
DELIMITER $$
CREATE TRIGGER before_employee_update
  BEFORE UPDATE ON employees
  FOR EACH ROW
BEGIN
```

```
    INSERT INTO employees_audit
    SET action = 'update',
     employeeNumber = OLD.employeeNumber,
       lastname = OLD.lastname,
       changedat = NOW();
END$$
DELIMITER ;
```

Inside the body of the trigger, we used the OLD keyword to access employeeNumber and lastn amecolumn of the row affected by the trigger.
Notice that in a trigger defined for INSERT, you can use NEW keyword only. You cannot use th e OLD keyword. However, in the trigger defined for DELETE, there is no new row so you can us e the OLD keyword only. In the UPDATE trigger, OLD refers to the row before it is updated and NEW refers to the row after it is updated.
Then, to view all triggers in the current database, you use SHOW TRIGGERS statement as follo ws:

```
1          SHOW TRIGGERS;
```

After that, update the employees table to check whether the trigger is invoked.

```
UPDATE employees
SET lastName = 'Khadilkar'
WHERE
   employeeNumber = 1056;
```

Finally, to check if the trigger was invoked by the UPDATE statement, you can query the emplo yees_audit table using the following query:

```
SELECT   *
FROM   employees_audit;
```

It inserted a new row into the employees_audit table.

Exception Handling

MySQL provides Handler to handle the exception in the stored procedure.

handler with examples:

```
1  /*Create Employee database for demo */
2  CREATE DATABASE Employee;
3  /*Create sample EmployeeDetails table.*/
4  CREATE TABLE Employee.tbl_EmployeeDetails
5  (
6     EmpID INTEGER
7    ,EmpName VARCHAR(50)
8    ,EmailAddress VARCHAR(50)
9    ,CONSTRAINT pk_tbl_EmployeeDetails_EmpID PRIMARY KEY (EmpID)
10 )ENGINE = InnoDB;
```

How to declare handler in store procedure:

Syntax of Handler:

```
1 DECLARE handler_action HANDLER FOR condition_value ... statement
```

Three type of Handler_Action:

- CONTINUE
- EXIT
- UNDO

Type of Condition Value:

- mysql_error_code
- sqlstate_value
- SQLWarning
- SQLException
- Not Found

How to write handler in stored procedure?

E.g.

```
1 DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SELECT 'Error occured';
2 DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET IsError=1;
3 DECLARE EXIT HANDLER FOR SQLEXCEPTION SET IsError=1;
4 DECLARE EXIT HANDLER FOR SQLSTATE '1418' SET IsError = 1;
```

These are four different handler examples.

Now, insert duplicate value into EmpID column

Description : This is demo stored procedure to
insert record into table with proper
error handling.

```
DELIMITER //
CREATE PROCEDURE Employee.usp_InsertEmployeeDetails
(
    InputEmpID INTEGER
   ,InputEmpName VARCHAR(50)
   ,InputEmailAddress VARCHAR(50)
)

BEGIN
   DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SELECT 'Error occured';
 INSERT INTO EmployeeDetails
 (
  EmpID
  ,EmpName
  ,EmailAddress
 )
 VALUES
```

```
(
 InputEmpID
 ,InputEmpName
 ,InputEmailAddress
);
 SELECT * FROM EmployeeDetails;
END //
DELIMITER ;
```

In the above Procedure, we defined a CONTINUE handler with custom exception message.

Now, call the above Procedure two times with same EmpID.

The first time, it will execute successfully, but the second time it will throw a custom error message.

As we defined CONTINUE handler, so it will just show an error message and CONTINUE to next part of the SELECT statement.

```
1 CALL Employee.usp_InsertEmployeeDetails (1,'Anvesh','anvesh@gmail.com');
2 CALL Employee.usp_InsertEmployeeDetails (1,'Roy','Roy@gmail.com');
```

Above are the two different calls with same EmpID value. The first call executes without any error message and the second call execute with an error message.

As we defined CONTINUE, so you can find two results in above examples. One is our custom error message and second is the result of the defined SELECT statement.
The execution didn't stop by error, and it continued for another part.

Now, check the EXIT handler:

Please modify your handler and replace CONTINUE by EXIT:

```
1  DELIMITER //
2  CREATE PROCEDURE Employee.usp_InsertEmployeeDetails
3  (
4      InputEmpID INTEGER
5     ,InputEmpName VARCHAR(50)
```

```
    ,InputEmailAddress VARCHAR(50)
)
BEGIN
   DECLARE EXIT HANDLER FOR SQLEXCEPTION SELECT 'Error occured';
INSERT INTO Employee.tbl_EmployeeDetails
(
 EmpID
 ,EmpName
 ,EmailAddress
)
VALUES
(
 InputEmpID
 ,InputEmpName
 ,InputEmailAddress
);
SELECT *FROM Employee.tbl_EmployeeDetails;
END
// DELIMITER ;
```

Call with the same parameter:

```
CALL Employee.usp_InsertEmployeeDetails (1,'Roy','Roy@gmail.com');
```

The Result is an only error message, and you cannot find two results as we defined EXIT to exit the code when an error occurred.

The best practice is to create a output parameter and store 1 if any error occurred.

Application code has to check this output parameter is NULL or 1.

1 = Error.

NULL = No Error.

Below is a stored procedure for this:

```
DELIMITER //
CREATE PROCEDURE Employee.usp_InsertEmployeeDetails

(
   InputEmpID INTEGER
  ,InputEmpName VARCHAR(50)
  ,InputEmailAddress VARCHAR(50)
  ,out IsError INTEGER
)
```

```
10 BEGIN
11    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET IsError=1;
12
13  INSERT INTO Employee.tbl_EmployeeDetails
14  (
15   EmpID
16   ,EmpName
17   ,EmailAddress
18  )
19  VALUES
20  (
21   InputEmpID
22   ,InputEmpName
23   ,InputEmailAddress
24  );
25  SELECT * FROM Employee.tbl_EmployeeDetails;
26 END
27 // DELIMITER ;
28
29
30
31
32
33
34
```

Now call the above Procedure and select output parameter:

```
CALL Employee.usp_InsertEmployeeDetails (1,'Roy','Roy@gmail.com',@IsError);
SELECT @IsError;
```

Some more examples of MySQL handler to handle exceptions or errors encountered in stored procedures.

When an error occurs inside a stored procedure, it is important to handle it appropriately, such as continuing or exiting the current code block's execution, and issuing a meaningful error message.

MySQL provides an easy way to define handlers that handle from general conditions such as warnings or exceptions to specific conditions e.g., specific error codes.

Declaring a handler

To declare a handler, you use the declare handler statement as follows:

```
DECLARE action HANDLER FOR condition_value statement;
```

If a condition whose value matches the condition value, MySQL will execute the statement and continue or exit the current code block based on the action Theaction accepts one of the following values:

- continue : the execution of the enclosing code block (`BEGIN` … `END`) continues.
- exit : the execution of the enclosing code block, where the handler is declared, terminates.

The condition_value specifies a particular condition or a class of conditions that activates the handler. The condition_value accepts one of the following values:

- A MySQL error code.
- A standard SQLSTATE value. Or it can be an SQLWARNING , NOTFOUND or SQLEXCEPTION condition, which is shorthand for the class of SQLSTATE values. The NOTFOUND condition is used for a cursor or SELECT INTO variable_list statement.
- A named condition associated with either a MySQL error code or SQLSTATE value.

The statement could be a simple statement or a compound statement enclosing by the BEGIN and END keywords.

MySQL error handling examples

Let's look into several examples of declaring handlers.

The following handler means that if an error occurs, set the value of the `has_error` variable to 1 and continue the execution.

```
1 DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET has_error = 1;
```

The following is another handler which means that in case any error occurs, rollback the previous operation, issue an error message, and exit the current code block. If you declare it inside the BEGIN END block of a stored procedure, it will terminate stored procedure immediately.

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
ROLLBACK;
SELECT 'An error has occurred, operation rollbacked and the stored procedure was terminated';
END;
```

The following handler means that if there are no more rows to fetch, in case of a cursor or SELECT INTO statement, set the value of the no_row_found variable to 1 and continue execution.

```
1 DECLARE CONTINUE HANDLER FOR NOT FOUND SET no_row_found = 1;
```

The following handler means that if a duplicate key error occurs, MySQL error 1062 is issued. It issues an error message and continues execution.

```
DECLARE CONTINUE HANDLER FOR 1062
SELECT 'Error, duplicate key occurred';
```

MySQL handler example in stored procedures

First, we create a new table named article_tags for the demonstration:

```
CREATE TABLE article_tags(
    article_id INT,
    tag_id    INT,
    PRIMARY KEY(article_id,tag_id)
);
```

The article_tags table stores the relationships between articles and tags. Each article may have many tags and vice versa. For the sake of simplicity, we don't create articles and tags tables, as well as the foreign keys in the article_tags table. Next, we create a stored procedure that inserts article id and tag id into the article_tags table:

```
DELIMITER $$
CREATE PROCEDURE insert_article_tags(IN article_id INT, IN tag_id INT)
BEGIN
DECLARE CONTINUE HANDLER FOR 1062
SELECT CONCAT('duplicate keys (',article_id,',',tag_id,') found') AS msg;

-- insert a new record into article_tags
INSERT INTO article_tags(article_id,tag_id)
VALUES(article_id,tag_id);

-- return tag count for the article
SELECT COUNT(*) FROM article_tags;
END
```

Then, we add tag id 1, 2 and 3 for the article 1 by calling the `insert_article_tags` stored procedure as follows:

```
CALL insert_article_tags(1,1);
CALL insert_article_tags(1,2);
CALL insert_article_tags(1,3);
```

After that, we try to insert a duplicate key to check if the handler is really invoked.

```
CALL insert_article_tags(1,3);
```

We got an error message. However, because we declared the handler as a CONTINUE handler, the stored procedure continued the execution. As the result, we got the tag count for the article as well.

If we change the CONTINUE in the handler declaration to EXIT , we will get an error message only.

```
DELIMITER $$

CREATE PROCEDURE insert_article_tags_2(IN article_id INT, IN tag_id INT)
BEGIN

 DECLARE EXIT HANDLER FOR SQLEXCEPTION
 SELECT 'SQLException invoked';

 DECLARE EXIT HANDLER FOR 1062
      SELECT 'MySQL error code 1062 invoked';

DECLARE EXIT HANDLER FOR SQLSTATE '23000'
SELECT 'SQLSTATE 23000 invoked';

-- insert a new record into article_tags
INSERT INTO article_tags(article_id,tag_id)
  VALUES(article_id,tag_id);

-- return tag count for the article
SELECT COUNT(*) FROM article_tags;
END
```

Finally, we can try to add a duplicate key to see the effect.

```
CALL insert_article_tags_2(1,3);
```

MySQL handler precedence

In case there are multiple handlers that are eligible for handling an error, MySQL will call the most specific handler to handle the error first.

An error always maps to one MySQL error code because in MySQL it is the most specific. An SQLSTATE may map to many MySQL error codes therefore it is less

specific. An SQLEXCPETION or an SQLWARNING is the shorthand for a class of SQLSTATES values so it is the most generic.
Based on the handler precedence's rules, MySQL error code handler, SQLSTATE handler and SQLEXCEPTION takes the first, second and third precedence.
Suppose we declare three handlers in the insert_article_tags_3 stored procedure as follows:

```
DELIMITER $$

CREATE PROCEDURE insert_article_tags_3(IN article_id INT, IN tag_id INT)
BEGIN

 DECLARE EXIT HANDLER FOR 1062 SELECT 'Duplicate keys error encounter
ed';
 DECLARE EXIT HANDLER FOR SQLEXCEPTION SELECT 'SQLException enc
ountered';
 DECLARE EXIT HANDLER FOR SQLSTATE '23000' SELECT 'SQLSTATE 230
00';

  -- insert a new record into article_tags
  INSERT INTO article_tags(article_id,tag_id)
  VALUES(article_id,tag_id);

  -- return tag count for the article
  SELECT COUNT(*) FROM article_tags;
END
```

We try to insert a duplicate key into the article_tags table by calling the stored procedure:

```
1 CALL insert_article_tags_3(1,3);
```

As you see the MySQL error code handler is called.

Using named error condition

Let's start with an error handler declaration.

```
DECLARE EXIT HANDLER FOR 1051 SELECT 'Please create table abc first';

SELECT * FROM abc;
```

What does the number 1051 really mean? Imagine you have a big stored procedure polluted with those numbers all over places; it will become a nightmare to maintain the code.

Fortunately, MySQL provides us with the DECLARE CONDITION statement that declares a named error condition, which associates with a condition.
The syntax of the DECLARE CONDITION statement is as follows:

```
1 DECLARE condition_name CONDITION FOR condition_value;
```

The condition_value can be a MySQL error code such as 1015 or a SQLSTATE value. The condition_value is represented by the condition_name . After declaration, we can refer to condition_name instead of condition_value .
So we can rewrite the code above as follows:

```
1 DECLARE table_not_found CONDITION for 1051;
2 DECLARE EXIT HANDLER FOR  table_not_found SELECT 'Please create table abc first';
3 SELECT * FROM abc;
```

This code is obviously more readable than the previous one.

Notice that the condition declaration must appear before handler or cursor declarations.

When an error occurs, the MySQL error code, SQLSTATE value, and message string are available using C API functions:

- MySQL error code: Call `mysql_errno()`
- SQLSTATE value: Call `mysql_sqlstate()`
- Error message: Call `mysql_error()`

User defined exception using signal(work only if version > 5.0)

This stored procedure would be more robust, and less likely to allow errors to slip by, if it actually raised an error condition when the date of birth was invalid. The ANSI SQL:2003 SIGNAL statement allows you to do this:

SIGNAL takes the following form:

```
SIGNAL SQLSTATE sqlstate_code|condition_name [SET MESSAGE_TEXT=string_or_variable];
```

You can create your own SQLSTATE codes (there are some rules for the numbers you are allo wed to use) or use an existing SQLSTATE code or named condition. When MySQL implements SIGNAL , you will probably be allowed to use a MySQL error code (within designated ranges) a s well.

When the SIGNAL statement is executed, a database error condition is raised that acts in exactl y the same way as an error that might be raised by an invalid SQL statement or a constraint vio lation. This error could be returned to the calling program or could be trapped by a handler in this or another stored program. If SIGNAL were available to us, we might write the employee d ate-of-birth birth procedure, as shown in Example 6-17.

```
CREATE PROCEDURE sp_update_employee_dob
   (p_employee_id int, p_dob date)
BEGIN
   DECLARE employee_is_too_young CONDITION FOR SQLSTATE '99001';

   IF DATE_SUB(curdate(), INTERVAL 16 YEAR) <P_DOB THEN
      SIGNAL employee_is_too_young
        SET MESSAGE_TEST='Employee must be 16 years or older';
   ELSE
     UPDATE employees
       SET date_of_birth=p_dob
      WHERE employee_id=p_employee_id;
   END IF;
END;
```

If we ran this new procedure from the MySQL command line (when MySQL implements SIGNA L), we would expect the following output:

```
mysql> CALL sp_update_employee(1,now());
ERROR 90001 (99001): Employee must be 16 years or older
```

Using SIGNAL , we could make it completely obvious to the user or calling program that the sto red program execution failed.

After trigger example

```
DELIMITER //


CREATE TRIGGER contacts_after_insert

AFTER INSERT

  ON contacts FOR EACH ROW
```

```
BEGIN

  DECLARE vUser varchar(50);

  -- Find username of person performing the INSERT into table
  SELECT USER() INTO vUser;

  -- Insert record into audit table
  INSERT INTO contacts_audit
  ( contact_id,
    created_date,
    created_by)
  VALUES
  ( NEW.contact_id,
    SYSDATE(),
    vUser );

END; //

DELIMITER ;
```

Rules for trigger

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

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

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

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

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

normalization rules

https://www.studytonight.com/dbms/boyce-codd-normal-form.php