MySQL

For Developers

Information Technology Institute

COURSE MATERIALS

You can access the course materials via this link

http://goo.gl/3R3eWh

DAY 3 CONTENTS

- Comparison functions.
- Control Flow functions
- Casting functions
- String functions
- Numeric functions
- Date/Time functions
- Stored Routines.
- Triggers
- Events
- Obtaining Metadata
- Backup and restore
- GUI Tools

BUILT IN FUNCTIONS

- Sometimes you need to modify and format your displayed result set in your query , this group of functions help in modifying and formatting such result set
- According to the input and output of those functions they can be classified into two main categories :
  - **Single Row** Functions <Scalar Functions> : work on a set of rows and return one row - Aggregation Functions.
  - **Multi Row** Functions : work on a set of rows in a row by row interaction mode fashion.

BUILT IN FUNCTIONS

- The Multi row functions are categorized according to the mode of action and argument`s data type into the following :
  - Comparison Functions
  - Control Flow Functions
  - Cast Functions
  - Managing Different Types of Data

COMPARISON FUNCTIONS

- These functions allow you to compare different values and, from those comparisons, return **one of the values** or return a condition of **true**, **false**, or NULL.
- If either argument or both arguments in a comparison are NULL, NULL is returned

GREATEST() AND LEAST()

- The values specified can be **numeric**, **string**, or **date/time** values and are compared based on the current character set
- Allow you to compare two or more values and return the value that is either the highest or lowest , depending on the function used.
- When you use this function, you must specify at least two values, although you can specify as many additional values as necessary.

```
SELECT GREATEST(4, 83, 0, 9, -3);-> 83

SELECT LEAST('cdef','ab','ghi'); -> ab
```

INTERVAL() AND STRCMP()

- The `INTERVAL()` takes only **integer expressions** as arguments.
- The value of the first argument is compared to the value of the subsequent arguments, and **returns the index of the last argument** that has a value that is **equal to or less than the first argument**:
- The `STRCMP()` returns 0 if the strings are the same, **-1** if the **first** argument is **smaller** than the second according to the current sort order, and 1 otherwise.

INTERVAL() AND STRCMP()

```
INTERVAL(2,5,7,8) → 0

INTERVAL(7,1,3,5,7,9) → 4

INTERVAL(10, 1, 2, 4, 8, 16) → 4;


STRCMP('a','a') → 0

STRCMP('a','A') → 0

STRCMP('A' COLLATE
latin1_general_ci,'a') → 0

STRCMP('A' COLLATE
latin1_general_cs,'a') → -1
```

ELT()

```
ELT(n,str1,str2,...)
```

- Returns the n-th string from the list of strings str1, str2, ... Returns NULL if n is NULL, the n-th string is NULL, or there is no n-th string.
- `ELT()` is complementary to `FIELD()`.

```
ELT(3,'a','b','c','d','e') → 'c'
ELT(0,'a','b','c','d','e') → NULL
ELT(6,'a','b','c','d','e') → NULL
ELT(FIELD('b','a','b','c'),'a','b','c') → 'b'
```

FIELD()

```
FIELD(arg0,arg1,arg2,...)
```

- Finds arg0 in the list of arguments arg1, arg2, ... and returns the index of the matching argument (beginning with 1). Returns 0 if there is no match or if arg0 is NULL.
- `FIELD()` is complementary to `ELT()`.

```
FIELD('b','a','b','c') → 2
FIELD('d','a','b','c') → 0
FIELD(NULL,'a','b','c') → 0
FIELD(ELT(2,'a','b','c'),'a','b','c')
→ 2
```

COALESCE() and ISNULL()

- The `COALESCE()` returns the **first value** in the list of arguments that **is not NULL**. If all values are NULL, then NULL is returned.

```
SELECT COALESCE(NULL, 2, NULL, 3); → 2
```

- The `ISNULL()` returns a value of **1** if the expression **evaluates to NULL**; otherwise, the function returns a value of **0**

```
SELECT ISNULL(1  *NULL); → 1
```

CONTROL FLOW FUNCTIONS

- Control flow functions enable you to choose between different values based on the result of an expression.
- Return a result by comparing conditions. The returned value is determined by which condition is **true** (non-zero, non-NULL).

IF()

```
IF(expr1,expr2,expr3)
```

- If expr1 is true (non-zero, non-NULL), returns expr2; otherwise, it returns expr3.

```
IF(1,'true','false') → 'true'

IF(0,'true','false') → 'false'

IF(NULL,'true','false') → 'false'

IF(1.3,'non-zero','zero') → 'non-zero'

IF(0.3 <> 0,'non-zero','zero') → 'non-zero'
```

IFNULL() AND NULLIF()

- `IFNULL` Returns expr2 if the value of the expression expr1 is NULL; otherwise, it returns expr1.

`SELECT IFNULL(NULL,'null')` → 'null'

`IFNULL('not null','null')` → 'not null‘

- `NULLIF` Returns expr1 if the two expression values differ, NULL if they are the same.

`SELECT NULLIF(10*20, 20*10);` → `NULL`

CASE()

- It has two forms of syntax:

```
CASE <expression>
WHEN <value> THEN <result>
[{WHEN <value> THEN <result>}...]
[ELSE <result>]
END

CASE WHEN condition THEN result
     [WHEN ...]
     [ELSE result]
END
```

CASE()

```
SELECT a,
       CASE a WHEN 1 THEN 'one'
              WHEN 2 THEN 'two'
              ELSE 'other'
       END
    FROM test;
```

CASE()

```
SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
    FROM test;
```

CAST FUNCTIONS

- Cast functions allow you to convert values to a specific type of data or to assign a character set to a value.
- The following are the casting functions:
  - `CAST();`
  - `CONVERT();`

```
SELECT CAST(20041031 AS DATE);

SELECT CONVERT(20041031, DATE);
```

MANAGING DATA TYPES

- According to the input data type they can be classified into :
  - String functions
  - Numeric functions
  - Date and Time functions

ASCII() AND ORD()

- The `ASCII()` function allows you to identify the numeric value of the first character in a string works only for single-byte characters (with values from 0 to 255).

```
SELECT ASCII(‘book’); → 98.
```

- The `ORD()` function works just like the `ASCII()` function except that it also supports multibyte characters.

```
SELECT ORD(37); → 51. ( for no. 3)
```

CHAR_LENGTH() AND LENGTH()

- The `CHAR_LENGTH()` and `CHARACTER_LENGTH()` functions, which are synonymous, return the **number** of characters in the specified string.

```
SELECT CHAR_LENGTH('cats and dogs'); → 13
```

- The `LENGTH()` function also returns the length of a string, only the length is measured in **bytes**, rather than characters.

```
SELECT LENGTH('cats and dogs'); → 13
```

CHARSET() AND COLLATION()

- The `CHARSET()` function identifies the character set used for a specified string.

```
SELECT CHARSET(‘cats and dogs’); → latin1
```

- The `COLLATION()` function identify the collation used for a string.

```
SELECT COLLATION(‘cats and dogs’); →
  latin1_swedish_ci
```

CONCAT() AND CONCAT_WS()

- They take on a number of string arguments and ‘glue’ them together into a new string.

```
SELECT CONCAT(‘cats’, ‘ ’, ‘and’, ‘ ’,
‘dogs’); # cats and dogs


SELECT CONCAT_WS(‘ ’, ‘cats’, ‘and’, ‘dogs’);
# cats and dogs
```

INSTR(), LOCATE() AND POSITION()

- Functions identifies where the substring is located in the string and returns the position number.

```
INSTR(<string>, <substring>)

SELECT INSTR(‘cats and dogs’, ‘dogs’); → 10


LOCATE(<substring>, <string>)

SELECT LOCATE(‘dogs’, ‘cats and dogs’); → 10


POSITION(<substring> IN <string>)

SELECT LOCATE(‘dogs’ IN ‘cats and dogs’); →
   10
```

FIND_IN_SET()

- `FIND_IN_SET()` returns the index of str within str_list. Returns 0 if str is not present in str_list, or NULL if either argument is NULL. The index of the first substring is 1.

`FIND_IN_SET('cow','moose,cow,pig')` → 2

`FIND_IN_SET('dog','moose,cow,pig')` → 0

SUBSTRING()

- The function, which includes several forms, returns a substring from the identified string.

```
SUBSTRING(<string>, <position>)

SUBSTRING(<string> FROM <position>)

SUBSTRING(<string> FROM <position> FOR <length>)


SELECT SUBSTRING(‘cats and dogs’, 10);

SELECT SUBSTRING(‘cats and dogs and more dogs’, 10, 4);
```

LOWER(), UPPER(), LEFT() AND RIGHT()

```
SELECT LOWER(‘Cats and Dogs’);  -> cats and dogs

SELECT UPPER(‘cats and dogs’);  -> CATS AND DOGS

SELECT LEFT(‘cats and dogs’, 4); -> cats

SELECT RIGHT(‘cats and dogs’,4); -> dogs
```

LTRIM(), RTRIM() AND TRIM()

```
SELECT LTRIM(‘ Cats and Dogs’);  -> Cats and Dogs

SELECT RTRIM(‘Cats and Dogs ’);  -> Cats and Dogs

SELECT TRIM(‘ Cats and Dogs ’);  -> Cats and Dogs
```

REPLACE() AND INSERT

```
SELECT REPLACE(‘Cats & Dogs’, ‘&’, ‘and’);
-> Cats and Dogs
```

- The **first** argument to `INSERT` is the string wherein the replacement is to take place. The **second** argument is the **offset** within the first argument where the newly inserted string will appear. The **third** argument is the number of characters of the first string that must be **overwritten**. Finally, the **last** argument is the string that will be placed into the first string argument at the specified position:

```
SELECT INSERT(‘Cats & Dogs’, 5, 3, ‘, Rats
and ’);  -> Cats, Rats and Dogs
```

REPEAT() AND REVERSE()

```
SELECT REPEAT(‘CatsDogs’, 3); →
  CatsDogsCatsDogsCatsDogs

SELECT REVERSE(‘bad’); → dab
```

ABS() AND SIGN()

- Numeric functions return NULL if you pass arguments that are out of range or otherwise invalid.

`ABS(x)`

- Returns the absolute value of x.

`ABS(13.5)` → 13.5

`ABS(-13.5)` → 13.5

`SIGN(-14.7)` → -1

TRIGONOMETRIC FUNCTIONS

`SIN(x)`

- Returns the sine of x, where x is measured in radians.

`SIN(0) → 0`

`SIN(PI()/2) → 1`

`COS(x)`

- Returns the cosine of x, where x is measured in radians.

`COS(0)` → 1

TRIGONOMETRIC FUNCTIONS

```
TAN(x)
```

- Returns the tangent of x, where x is measured in radians.

```
TAN(0)  → 0
TAN(PI()/4)  → 1
```

```
COT(x)
```

- Returns the cotangent of x, where x is measured in radians.

```
COT(PI()/4)  → 1
```

TRUNCATE()

```
TRUNCATE(x,d)
```

- Returns the value x, with the fractional part truncated to d decimal places. If d is 0, the result has no decimal point or fractional part. If d is greater than the number of decimal places in x, the fractional part is right-padded with trailing zeros to the desired width.

```
TRUNCATE(1.23,1) → 1.2
```

```
TRUNCATE(1.23,0) → 1
```

```
TRUNCATE(1.23,4) → 1.2300
```

CEILING()

`CEILING(x)`

`CEIL(x)`

- Returns the smallest integer not less than x. If the argument has an exact-value numeric type, the return value does, too. Otherwise the return value has a floating point (approximate-value) type. This is true even though the value has no fractional part.

`CEILING(3.8)` → 4

`CEILING(-3.8)` → -3

FLOOR()

`FLOOR(x)`

- Returns the largest integer not greater than x. If the argument has an exact-value numeric type, the return value does, too. Otherwise the return value has a floatingpoint (approximate-value) type. This is true even though the value has no fractional part.

`FLOOR(3.8)` → 3

`FLOOR(-3.8)` → -4

ROUND(),SQRT(), MOD(), POW() and PI()

```
SELECT ROUND(4.27943, 2); → 4.28

SELECT SQRT(36); → 6

SELECT MOD(22, 7); → 1

SELECT POW(4, 2); → 16

SELECT PI() → 3.141593
```

ADDDATE() and DATE_ADD()

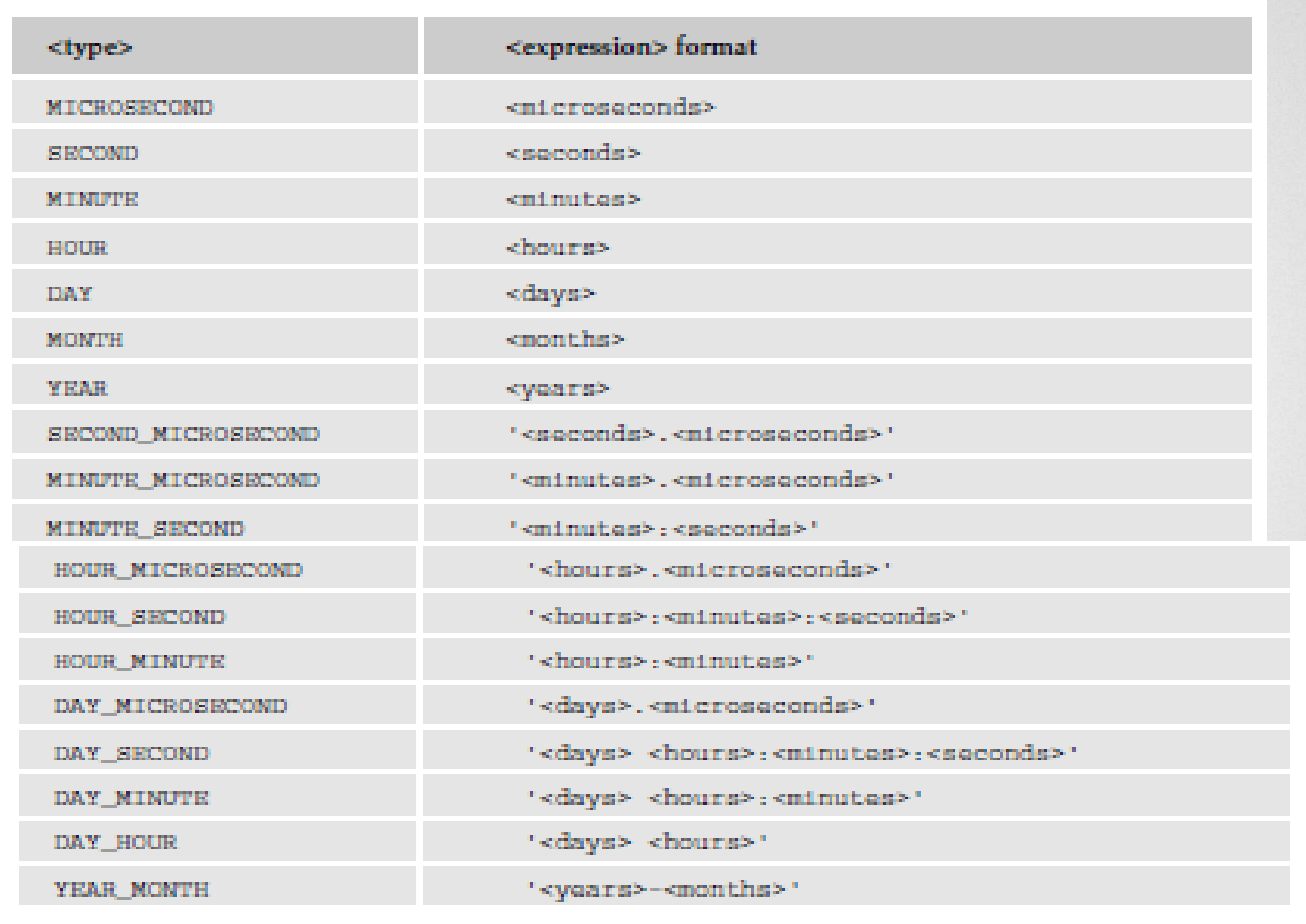
- The `ADDDATE()` and `DATE_ADD()` functions, which are synonymous, allow you to add date-related intervals to your date values.

```
ADDDATE(<date>, <days>)

SELECT ADDDATE(‘2004-11-30 23:59:59’, 31);
```

```
ADDDATE(<date>, INTERVAL <expression> <type>)

SELECT ADDDATE(‘2004-10-31 13:39:59’,
  INTERVAL ‘10:20’ HOUR_MINUTE);
```

- The following table lists the types that you can specify in the `INTERVAL` clause and the format for the expression used with that type:

ADDDATE() and DATE_ADD()



<type>	<expression> format
MICROSECOND	<microseconds>
SECOND	<seconds>
MINUTE	<minutes>
HOUR	<hours>
DAY	<days>
MONTH	<months>
YEAR	<years>
SECOND_MICROSECOND	'<seconds>.<microseconds>'
MINUTE_MICROSECOND	'<minutes>.<microseconds>'
MINUTE_SECOND	'<minutes>:<seconds>'
HOUR_MICROSECOND	'<hours>.<microseconds>'
HOUR_SECOND	'<hours>:<minutes>:<seconds>'
HOUR_MINUTE	'<hours>:<minutes>'
DAY_MICROSECOND	'<days>.<microseconds>'
DAY_SECOND	'<days> <hours>:<minutes>:<seconds>'
DAY_MINUTE	'<days> <hours>:<minutes>'
DAY_HOUR	'<days> <hours>'
YEAR_MONTH	'<years>-<months>'

SUBDATE() and EXTRACT()

```
SUBDATE(<date>, INTERVAL <expression> <type>)

SELECT SUBDATE(‘2004-10-31 23:59:59’,
  INTERVAL ‘12:10’ HOUR_MINUTE);
```

```
SUBDATE(<date>, <days>)

SELECT SUBDATE(‘2004-12-31 23:59:59’, 31);
```

```
EXTRACT(<type> FROM <date>)

SELECT EXTRACT(YEAR_MONTH FROM ‘2004-12-31
  23:59:59’);
```

CURDATE(), CURTIME() & NOW()

- The `CURDATE()`, `CURTIME()`, and `NOW()` functions are particularly useful if you need to insert a date in a column that is based on the current date or time.

```
SELECT CURDATE(); → 2004-09-08

SELECT CURTIME(); → 16:07:46

SELECT NOW(); → 2004-09-08 16:08:00
```

DATE(), MONTH(), MONTHNAME() &YEAR()

The `DATE()`, `MONTH()`, `MONTHNAME()`, and `YEAR()` functions are helpful when you want to retrieve a portion of a date or a related value based on the date and use it in your application.

```
SELECT DATE(‘2004-12-31 23:59:59’); → 2004-12-31

SELECT MONTH(‘2004-12-31 23:59:59’);  → 12

SELECT MONTHNAME(‘2004-12-31 23:59:59’); → December

SELECT YEAR(‘2004-12-31 23:59:59’); → 2004.
```

DATEDIFF() & TIMEDIFF()

- The `DATEDIFF()` and `TIMEDIFF()` functions are useful when you have a table that includes two time/date columns.

```
SELECT DATEDIFF(‘2004-12-31 23:59:59’, ‘2003-12-31 23:59:59’); → 366 (because 2004 is a leap year).

SELECT TIMEDIFF(‘2004-12-31 23:59:59’, ‘2004-12-30 23:59:59’); → 24:00:00
```

DAY(), DAYNAME(), DAYOFWEEK(), and DAYOFYEAR()

- The `DAY()`, `DAYOFMONTH()`, `DAYNAME()`, `DAYOFWEEK()`, and `DAYOFYEAR()` functions can be useful if you want to extract specific types of information from a date.

```
SELECT DAY(‘2004-12-31 23:59:59’); → 31

SELECT DAYNAME(‘2004-12-31 23:59:59’); →
Friday

SELECT DAYOFWEEK(‘2004-12-31 23:59:59’); → 6

SELECT DAYOFYEAR(‘2004-12-31 23:59:59’); →
366
```

SECOND(), MINUTE(), HOUR(), and TIME()

```
SELECT SECOND('2004-12-31 23:59:59'); → 59

SELECT MINUTE('2004-12-31 23:59:59'); → 59

SELECT HOUR('2004-12-31 23:59:59'); → 23

SELECT TIME('2004-12-31 23:59:59'); →
  23:59:59
```

STORED ROUTINES

- A **stored routine** is a set of SQL statements that can be stored in the server. There are two types of stored routines:
  - **Stored Procedures** – A series of instructions stored within the database itself that acts upon the instructions but does not return a value. A procedure is invoked using a `CALL` statement, and can only pass back values using output variables.
  - **Stored Functions** – A series of instructions stored within the database itself that returns a single value. A function can be called from inside a statement just like any other function (that is, by invoking the function's name), and can return a scalar value.

ADVANTAGES OF STORED ROUTINES

- **Different Client Applications** – Stored routines give developers the ability to create a statement in one application (MySQL) that can be utilized in multiple client applications.
- **Security** – Stored routines provide an encapsulation for SQL statements.
- **Performance** – Stored routines provide improved performance because less information needs to be sent between the server and the client

DISADVANTAGES OF STORED ROUTINES

- **Increased Server Load** - Executing stored routines in the database itself can increase the server load and reduce the performance of the applications
- **Limited Development Tools** - There are currently a limited number of development tools to support stored routines in MySQL.
- **Limited Language Functionality and Speed** - Even though having logic in the database itself is a huge advantage in many situations, there is definitely limitations on what can be accomplished in comparison to other programming languages.
- **Limited Debugging/Profiling Capabilities**

STORED FUNCTIONS CREATION

```
CREATE FUNCTION function_name
returns data_type
return function_statement
```

- Example of a stored function that has one SQL statement

```
CREATE FUNCTION world_record_count ()
RETURNS INT
RETURN SELECT COUNT(*) FROM Country;
```

- The stored function is invoked by:

```
SELECT function_name();
```

STORED PROCEDURE CREATION

```
CREATE PROCEDURE procedure_name
procedure_statement
```

- Example of a stored procedure that has one SQL statement

```
CREATE PROCEDURE world_record_count ()

SELECT 'country count ', COUNT(*) FROM
Country;
```

- The stored procedure is invoked by:

```
CALL procedure_name();
```

COMPOUND STATEMENT

- Example of a stored procedure that has more than one SQL statement

```
DELIMITER //

CREATE PROCEDURE world_record_count ()

BEGIN

SELECT 'country count ', COUNT(*) FROM country;

SELECT 'city count ', COUNT(*) FROM city;
SELECT 'CountryLanguage count', COUNT(*) FROM CountryLanguage;

END//

DELIMITER ;
```

VARIABLES IN STORED PROCEDURES

- Variables are used in stored procedure to store the immediate result.

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

- The **variable name** should follow the naming convention and should not be the same name of table or column in a database
- The **data type** of the variable, it can be any primitive type which MySQL supports such as `INT`, `VARCHAR` and `DATETIME`...along with the data type is the size of the variable. When you declare a variable, its initial value is `NULL`.

VARIABLES IN STORED PROCEDURES

- You can also assign the **default value** for the variable by using DEFAULT statement.

```
DECLARE total_sale INT DEFAULT 0

DECLARE x, y INT DEFAULT 0
```

- A variable with the '@' at the beginning is Session Variable. It exists until the session end.
- Variables scope:
  - A variable has its own scope.
  - If you declare a variable inside a stored procedure, it will be out of scope when the `END` of stored procedure reached.
  - You can declare two variables or more variables with the same name in different scopes; the variable only is effective in its scope.

ASSIGNING VARIABLES

- Once you declared a variable, you can start using it. To assign other value to a variable you can use
  - `SET` statement
  - `SELECT … INTO` to assign a query result to a variable.

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

```
DECLARE total_products INT DEFAULT 0;
SELECT COUNT(*) INTO total_products FROM
products;
```

PARAMETER DECLARATIONS

```
MODE param_name param_type(param_size)
```

- Almost stored procedures you develop require parameters. Parameters make the stored procedure more flexible and useful.
- **param_name** is the name of the parameter. The name **must not** be the same as the column name of tables and following naming convention.
- Each parameter is separated by a comma if the stored procedure more than one parameter.

PARAMETER DECLARATIONS

- **MODE** could be IN, OUT or INOUT depending on the purpose of parameter you specified:
  - IN this is the default mode. IN indicates that a parameter can be passed into stored procedures but any modification inside stored procedure does not change parameter.
  - OUT this mode indicates that stored procedure can change this parameter and pass back to the calling program.
  - INOUT obviously this mode is combined of IN and OUT mode; you can pass parameter into stored procedure and get it back with the new value from calling program.

PARAMETER DECLARATIONS

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


CALL GetOfficeByCountry('USA')
```

PARAMETER DECLARATIONS

```
DELIMITER $$
 CREATE PROCEDURE CountOrderByStatus
(IN orderStatus VARCHAR(25), OUT var INT)
 BEGIN
    SELECT count(orderNumber) INTO var
   FROM orders
   WHERE status = orderStatus;
END$$
DELIMITER ;
CALL CountOrderByStatus('Shipped',@total);
SELECT @total AS total_shipped;
```

CONTROL FLOW

- The two common flow controls are:
  - **Choices** – statements that are obeyed under certain conditions. In MySQL, these are represented in the **IF** and **CASE** statements.
  - **Loops** - statements that are obeyed repeatedly. In MySQL these are represented in the **REPEAT**, **WHILE** and **LOOP** statements

IF

```
IF (test_condition)
THEN ...
ELSEIF (test_condition)
THEN ...
ELSE ...
END IF
```

CASE

```
CASE case_value

WHEN when_value

THEN ...

ELSE ...

END CASE
```

```
CASE WHEN test_condition

THEN ...

ELSE ...

END CASE
```

REPEAT

```
mylabel: REPEAT
...
UNTIL test_condition
END REPEAT mylabel
```

REPEAT

```
DELIMITER $$
DROP PROCEDURE IF EXISTS RepeatLoopProc$$
CREATE PROCEDURE RepeatLoopProc()
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 ;
```

WHILE

```
mylabel: WHILE test_condition DO

...

END WHILE mylabel
```

WHILE

```
DELIMITER $$
DROP PROCEDURE IF EXISTS WhileLoopProc$$
CREATE PROCEDURE WhileLoopProc()
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 ;
```

LOOP

```
mylabel: LOOP
...
[LEAVE | ITERATE ] mylabel;
END LOOP mylabel
```

LOOP

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

DECLARE HANDLER

- The `DECLARE … HANDLER` statement specifies handlers that can deal with, one or more conditions.
- The condition can be handled in one of two possible ways:
  - `CONTINUE –` execution of the current stored routine will continue after the statement that caused the handler to be activated.
  - `EXIT` – terminates the current `BEGIN … END` compound statement (block) where the handler was declared

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '23000'
SET @x = 1;
```

DECLARE HANDLER

```
DELIMITER //
CREATE PROCEDURE dohandler ()
BEGIN
DECLARE dup_keys CONDITION FOR SQLSTATE '23000';
DECLARE CONTINUE HANDLER FOR dup_keys SET
@garbage = 1;
SET @x = 1;
INSERT INTO world.d_table VALUES (1);
SET @x = 2;
INSERT INTO world.d_table VALUES (1);
SET @x = 3;
END//
DELIMITER ;
```

CURSORS

- Cursors are a control structure within stored routines that are for the **retrieval of records**, **one row at a time**. The term “cursor” is short for CURrent Set Of Records

```
DECLARE cursor_name CURSOR FOR
select_statement;
```

- Cursors **must** be declared **before** declaring handlers. Note: **Variables** along with conditions must be declared **before** declaring either cursors or handlers.
- There is a need to identify when the set of records being “retrieved” has reached it's end (SQLSTATE '02000').

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'
SET done = 'yes'
```

CURSORS

- Then to use the CURSOR,

```
OPEN cursor_name;

FETCH cursor_name INTO var_name1, var_name2,
var_name3, ... ;

CLOSE cursor_name;
```

CURSORS

```
DELIMITER $$
CREATE PROCEDURE `test`()
BEGIN
DECLARE done BOOL DEFAULT FALSE;
DECLARE i INTEGER;
DECLARE n TEXT;
DECLARE curs1 CURSOR FOR SELECT `id`, `name` FROM
test_tbl;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done =
TRUE;
OPEN curs1;
read_loop: LOOP
IF done THEN
LEAVE read_loop;
END IF;

FETCH curs1 INTO i,n;
SELECT i,n;
END LOOP;
CLOSE curs1;
END$$
DELIMITER ;
```

EXAMINE STORED ROUTINES

```
SHOW CREATE PROCEDURE proc_name;

SHOW CREATE FUNCTION func_name;
```

- To list the stored routines associated with a database:

```
SHOW PROCEDURE STATUS where Db=‘db_name’;
SHOW FUNCTION STATUS where Db=‘db_name’;
```

- To drop a stored routines :

```
DROP PROCEDURE [IF EXISTS] proc_name;

DROP FUNCTION [IF EXISTS] func_name;
```

TRIGGERS

- SQL trigger is a stored SQL statements that can be activated or fired when an event associated with a table occurs.
- The event can be any event including `INSERT`, `UPDATE` and `DELETE`.
- Triggers are useful for:
  - Logging functionality
  - Examining data before it is inserted or updated, or verify deletes or updates
  - Mimicking the behavior of foreign keys for storage engines that do not support foreign keys.

TRIGGER CREATION

- To create a trigger, You can use this form.

```
CREATE TRIGGER trigger_name

{ BEFORE | AFTER }

{ INSERT | UPDATE | DELETE }

ON table_name

FOR EACH ROW

triggered_statement
```

- It’s recommended to name your trigger like the following:

```
(BEFORE | AFTER)_tableName_(INSERT| UPDATE |
  DELETE)
```

TRIGGER CREATION

```
CREATE TRIGGER After_City_Delete AFTER DELETE
ON City

FOR EACH ROW

INSERT INTO DeletedCity (ID, Name) VALUES
(OLD.ID, OLD.Name);
```

TRIGGER CREATION

```
CREATE TABLE student_audit
( id int(11) NOT NULL AUTO_INCREMENT,
studentNumber int(11) NOT NULL,
name varchar(50) NOT NULL,
changedon datetime DEFAULT NULL,
action varchar(50) DEFAULT NULL,
PRIMARY KEY (id)  );
```

TRIGGER CREATION

```
DELIMITER $$

CREATE TRIGGER before_student_update  BEFORE
  UPDATE ON student

FOR EACH ROW BEGIN

INSERT INTO student_audit

SET action = 'update',

studentNumber = OLD.studentNumber,

name = OLD.name,

changedon = NOW();

END$$

DELIMITER ;
```

TRIGGERS

- While trigger is implemented in MySQL has all features in standard SQL but there are some restrictions you should be aware of like following:
  - It is not allowed to call a stored procedure in a trigger.
  - It is not allowed to create a trigger for views or temporary table.
  - Creating a trigger for a database table causes the query cache invalidated.
  - All triggers for a table must have unique name. It is allowed that triggers for different tables having the same name but it is recommended that trigger should have unique name in a specific database.

EXAMINE TRIGGERS

- To remove a trigger, You can use this form.

```
DROP TRIGGER trigger_name;
```

- To list the triggers within a database:

```
SHOW TRIGGERS FROM  db_name;
```

EVENTS

- MySQL 5.1.6 and up has an event scheduler that enables you to perform time-activated database operations.
- An **event** is a stored program that is associated with a schedule. The schedule defines the time or times at which the event executes, and optionally when the event ceases to exist
- Events are especially useful for performing unattended administrative operations such as periodic updates to summary reports, expiration of old data, or log.
- To turn on event scheduler (process that runs in the background and constantly looks for events to execute)

```
SET GLOBAL event_scheduler = ON;
```

EVENT CREATION

- General form

```
CREATE EVENT [IF NOT EXISTS] event_name
    ON SCHEDULE schedule
    DO event_body;

schedule:
    AT timestamp [+ INTERVAL interval] ...
  | EVERY interval
    STARTS timestamp ENDS timestamp
interval:
    quantity {YEAR | QUARTER | MONTH | DAY | HOUR | MINUTE |
              WEEK | SECOND | YEAR_MONTH | DAY_HOUR | DAY_MINUTE |
              DAY_SECOND | HOUR_MINUTE | HOUR_SECOND |
MINUTE_SECOND}
```

EVENT CREATION

- Suppose that you have a table named `web_session` that holds state information for sessions associated with users who visit your Web site, and that this table has a `DATETIME` column named `last_visit` that indicates the time of each user’s most recent visit.
- To keep this table from accumulating stale rows, you can set up an event that periodically purges them.
- To execute the event every six hours and have it expire rows more than a day old, write the event definition like this:

EVENT CREATION

```
CREATE EVENT expire_web_session

ON SCHEDULE EVERY 4 HOUR

DO

DELETE FROM web_session

WHERE last_visit < CURRENT_TIMESTAMP -
INTERVAL 1 DAY;
```

EVENT OPERATIONS

- To remove an event, You can use this form.

```
DROP EVENT [IF EXISTS] event_name;
```

- To modify an event, You can use this form.

```
ALTER EVENT event_name RENAME TO new_event;
ALTER EVENT event_name DO
Event_stamnts
```

COMMENTS

- MySQL supports **three** forms of comment syntax.
- One of those forms has variants that allow special instructions to be passed to the MySQL server.

1. C-style comment .

```
/* this is a comment */

/*

this is a comment,

spanning multiple lines

*/

SHOW /*!50002 FULL */ TABLES;
```

COMMENTS

2. A -- (double dash) sequence followed by a space .

```
-- This is a comment
```

3. A # character begins a comment that extends to the end of the line

```
# This is also a comment
```

OBTAINING METADATA

- You can think of `INFORMATION_SCHEMA` as a virtual database in which the tables are views for different kinds of database metadata.
- To see what tables `INFORMATION_SCHEMA` contains, use `SHOW TABLES`.
- The following list briefly describes the `INFORMATION_SCHEMA` tables just shown:

```
 SCHEMATA, TABLES, VIEWS, ROUTINES, TRIGGERS,
EVENTS, PARTITIONS, COLUMNS
```

- As you see, it contains information about databases; tables, views, stored routines, triggers, and events within databases; table partitions; and columns within tables

OBTAINING METADATA

- Show table engine

```
SELECT ENGINE FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_SCHEMA=‘grade_book' AND
TABLE_NAME='student';
```

- Show Variables:

```
SHOW VARIABLES;

SHOW SESSION VARIABLES;
```

- Show Warnings:

```
SHOW WARNINGS;
```

OBTAINING METADATA

```
SHOW GRANTS;

SHOW GRANTS FOR CURRENT_USER();

SHOW GRANTS FOR ‘islam'@'localhost';


SET PASSWORD FOR ‘islam'@'localhost' =
PASSWORD(‘os123');
```

CHECKING STATUS VARIABLE VALUES

- The server maintains status variables that enable you to monitor its runtime operation. You can display these variables:

```
SHOW STATUS;

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

| Variable_name            | Value  |

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

| Aborted_clients          | 0      |

| Aborted_connects         | 1      |

| Binlog_cache_disk_use    | 0      |

….
```

BACKUP DATABASE INTO FILE

- Save/Restore data using select

```
SELECT * INTO OUTFILE ‘~/city' FROM City;
LOAD DATA INFILE ‘~/city' INTO TABLE City;
```

- Create a dump file using:

```
mysqldump --databases sampdb > sampdb.sql
```

- load the dump file into the MySQL

```
mysql < ~/sampdb.sql
mysqlimpot –u root –p sampdb.sql
```

GUI TOOLS