MySQL PL (programming language):-

- product of MySQL
- MySQL programming language
- programming language of MySQL
- used for database programming
- e. g. HRA_CALC, TAX_CALC, ATTENDACNCE_CALC, etc,
- used for server side data processing
- can be called through any front-end software
- e. g. MySQL Command Line Client, MySQL Workbench, JAVA, MS .NET, etc.
- support few 4 GL features
- **Block level language**

Benefits of block level language :-

1. Modularity
2. control scope of variables (form of Encapsulation) (form of data hiding)
3. Efficient error management (localize the error in the case of Exceptions)

- screen input and screen output is not allowed (e. g. scanf, printf, etc. not available)
- **USED ONLY FOR PROCESSING**
- you can select the statement inside the block but it is not recommended
- SQL commands that are allowed inside MySQL PL :

  DDL, DML, DQL, DTL / TCL.

- DCL command not allowed inside the block
- syntax :

```
BEGIN
-------
BLOCK OF CODE
-------
END;


CREATE TABLE TEMPP (

FIR INT(4),
SEC CHAR(15)
);
```

STORED PROCEDURES

- stored object
- objects that are stored in the database
- MySQL PL programs are written in the form of stored procedures
- Routine (set of command) that has to be called explicitly (procedure is a void function)
- global procedures
- can be called through MySQL command line client, MySQL Workbench, JAVA, MS .NET, etc.
- can be called through any front-end software

- stored in the database in compiled format
- hence the execution will be very fast
- hiding source code from end user
- In multi-user environment, if multiple users are calling the same stored procedure simultaneously, then only a single copy of the procedure code is brought into server RAM, the procedure code will be shared by all the users.
- procedure can have local variable
- inside the procedure, you can have any processing (full MySQL / PL is allowed)
- one procedure can call other procedure
- procedure can call itself (known as recursion)
- you can pass parameter to a procedure
- Overloading of stored procedure is not allowed

Create procedure

```
delimiter //
create procedure hello()
begin
insert into tempp values (1,'hello');
end; //
delimiter ;

; ---- > is known as terminator


## delimiter is used to make separate block

delimiter //

-------

// delimiter
```

Call To Procedure :-

```
call < name of procedure >;
```

Drop Procedure :-

```
Drop procedure < name of procedure >;
```

Using variable

- in MySQL, when you declare a variable, if you don't initialize it, then it will store a null value

```
DELIMITER $$
CREATE PROCEDURE using_variable()
BEGIN
declare x int(4);
set x =10;
insert into TEMP values(x,'atul');
END; $$
DELIMITER ;
```

Using Default Variable

```
DELIMITER $$
CREATE PROCEDURE using_default_variable()
BEGIN
declare x int(4) default 10;
declare y char(6) default 'sakshi';
insert into TEMP values(x,y);
END; $$
DELIMITER ;
```

Using parameter passing

```
DELIMITER $$
CREATE PROCEDURE using_default_variable( x int(4), y float(2,1))
BEGIN
    declare hra float(7,2);
    set hra = x*y;
insert into TEMP values(hra,'HRA');
END; $$
DELIMITER ;

call using_default_variable(10,0.4);
```

Store the value from table to the variable

- in this case value sal column are copy into x where ename is 'D'.

```
DELIMITER $$
CREATE PROCEDURE store_table_into_variable( )
BEGIN
    declare x int(4);
    declare y varchar(4);
    select sal,ename into x, y from emp where ename = 'E';
insert into TEMP values(x,y);
END; $$
DELIMITER ;
```

To see which all procedures you created

```
show procedure status;

show procedure status where DB = 'Atul'

show procedure status where name like 'A%';
```

To see the source code of stored procedures

```
show create procedure abc;
```

IF STATEMENT (used for decision making)

```
DELIMITER $$
CREATE PROCEDURE if_statements( )
BEGIN
    declare x int(4);
    select sal into x from emp where ename = 'E';

    if x > 400 then
insert into TEMP values(x,'high sal');
else
if x < 400 then
insert into TEMP values(x,'low sal');
else
        insert into TEMP values(x,'low sal');
        end if;
end if;

END; $$
DELIMITER ;

DELIMITER $$
CREATE PROCEDURE if_statements( )
BEGIN
    declare x boolean default TRUE;
    if x then
insert into TEMP values(x,'mumbai');
end if;
END; $$
DELIMITER ;
```

CASE STATEMENTS

CASE statement is faster than IF statement, but **IF** more powerful than **CASE** (nesting id possible with **IF** but not supported by **CASE**)

in the case statement ELSE is optional

if ELSE is not provided and if none of the cases are satisfied then MySQL will give an error message

if you want to skip the else and avoid the error message then you supply ELSE with an empty Begin and END block

```
DELIMITER $$
CREATE PROCEDURE case_statements( )
BEGIN
    declare x int(4);
    select sal into x from emp where ename = 'C';

    case
    when x > 4000 then
    insert into values(x,'High sal');
    when x < 4000 then
    insert into values(x,'low sal');
    else
    insert into values(x,'medium sal');
    end case;
END; $$
DELIMITER ;
DELIMITER $$
CREATE PROCEDURE case_statements( )
BEGIN
    declare x int(4);
    select sal into x from emp where ename = 'C';

    case
    when x > 4000 then
    insert into values(x,'High sal');
    when x < 4000 then
    insert into values(x,'low sal');
    else
    insert into values(x,'medium sal');
    end case;
END; $$
DELIMITER ;
```

WHILE LOOP

```
DELIMITER $$
CREATE PROCEDURE while_loop( )
BEGIN
    declare x int(4) default 1;
    declare y int(4) default 1;

    while x < 10 do
    while y < 10 do
    insert into temp value (y, 'in y loop');
    set y = y+1;
    end while;
        insert into temp values(x, 'in x loop');
        set x = x+1;
    end while;

END; $$
DELIMITER ;
```

REPEAT LOOP

there is no criteria to enter the loop, but there is a condition to end the loop

it will execute at least once like do while loop

```
DELIMITER $$

CREATE PROCEDURE repeat_loop( )

BEGIN

     declare x int(4) default 1;
```

```
    repeat
                                insert into temp value (x, 'repeat loop');
                                set x = x+1;
           until x > 5
     end repeat;

END; $$
DELIMITER ;
```

LOOP, LEAVE, AND ITERATE STATEMENTS:-

LEAVE

- LEAVE statement allows you to exit the loop (similar to break statement of 'c' programming)

ITERATION

- INTERATION statement allows you to skip the entire code under it and start a new iteration (similar to continue statement of c).

LOOP

- LOOP statement execute a block of code repeatedly with an additional flexibility of using a loop label

Example for all of this

```
DELIMITER $$
CREATE PROCEDURE loop_leave_iterate( )
BEGIN
                    declare x int(4) default 1;
                    pqr_loop : loop
                                        if x > 10 then
                                                  leave pqr_loop;
                                        end if;
            set x = x+1;
            if mod(x,2) != 0 then
                              iterate pqr_loop;
            else
```

```
                                          insert into temp values(x, 'inside loop');

                      end if;

           end loop;

END; $$

DELIMITER ;
```

SESSION VARIABLE

```
set @x = 10 ;

select @x from dual;
```

CURSORS

- cursors are present in all RDBMS
- cursors are used to locking the rows manually
- cursors are present in some DBMS (e. g. MS Excel, FoxPro, etc.)
- cursors are present in some front-ends also (e. g. MS .NET, PowerBuilder, etc.)
- cursors is a type of a variable
- cursor is work with the rowid
- cursor can store multiple rows
- used for storing multiple rows
- used for handling multiple rows
- used for processing multiple rows
- used for storing the data temporarily
- similar to 2D array
- cursor is based on SELECT statement
- Cursor is a READ_ONLY variable
- the data that is present in the cursor, it cannot be manipulated
- you will have to fetch 1 row at a time into some intermediate variable and do your processing with those variables.
- you can only fetch sequentially (top to bottom)
- you cannot fetch backwards in MySQL Cursors
- you can only fetch 1 row a time
- Example :-

```
DELIMITER $$
CREATE PROCEDURE cursor1( )
BEGIN
                declare a int(4);
                declare b varchar(15);
                declare c int(4);
```

```
                declare d int(2);
```

```
                        declare x int(4) default 1;

                        declare c1 cursor for select * from emp;

                        open c1;    ----- >  open cursor

                        open c1;    ----- >  error ( cursor is already open )

                        while x < 6 do

                                    fetch c1 into a,b,c,d;

                                    /* processing, e. g. set hra = c*0.4, etc. */

                                    insert into temp values(a,b);

                                    set x = x+1;

                        end while;

                        close c1;

END; $$

DELIMITER ;
```

you will have to close cursor before open new cursor

- **To reset the cursor pointer**

  close c1; open c1;

```
 DELIMITER $$

CREATE PROCEDURE cursor1( )

BEGIN

                        declare a int(4);

      declare b varchar(15);

      declare c int(4);

      declare d int(2);

      declare x int(4) default 1;

      declare c1 cursor for select * from emp;

                         open c1;    ----- >  open cursor


             open c1;    ----- >  error ( cursor is already open )


             while x < 11 do  ------- >   it will throw error ( number of rows are less )


                                    fetch c1 into a,b,c,d;

                                    /* processing, e. g. set hra = c*0.4, etc. */
```

```
                                        insert into temp values(a,b);
                                        set x = x+1;

    end while;
    close c1;
END; $$
DELIMITER ;





##### Automatic count rows

DELIMITER $$
CREATE PROCEDURE cursor3( )
BEGIN
                          declare a int(4);
      declare b varchar(15);
      declare c int(4);
      declare d int(2);
      declare x int(4) default 1;
      declare y int(2);
      declare c1 cursor for select * from emp;

    select count(*) into y from emp;

    open c1;

    while x < y do
                                        fetch c1 into a,b,c,d;
                                        /* processing, e. g. set hra = c*0.4, etc. */
                                        insert into temp values(a,b);
                                        set x = x+1;
    end while;
    close c1;
```

```
END; $$

DELIMITER ;
```

Continue handler

- not found is a cursor attribute, it returns a Boolean TRUE value if the last fetch was unsuccessful, and a FALSE value if the last fetch was successful
- when the fetch is unsuccessful, the not found cursor attribute returns a Boolean TRUE value then it will raise the continue handler the continue handler when raised will be set finished variable = 1
- the continue handler has to be declare after the cursor declaration

```
DELIMITER $$

CREATE PROCEDURE cursor4( )

BEGIN

                        declare a int(4);

    declare b varchar(15);

    declare c int(4);

    declare d int(2);

    declare finished int (4) default 0;

    declare c1 cursor for select * from emp;

    declare continue handler for not found set finished = 1;

DELIMITER $$

CREATE PROCEDURE cursor4( )

BEGIN

                        declare a int(4);

    declare b varchar(15);

    declare c int(4);

    declare d int(2);

    declare finished int (4) default 0;

    declare c1 cursor for select * from emp;

    declare continue handler for not found set finished = 1;

      open c1;

  cursor_c1_loop:loop

                                fetch c1 into a,b,c,d;

                                if finished = 1 then

                                        leave cursor_c1_loop;

                                end if;
```

```
                                              insert into temp values(a,b);
    end loop cursor_c1_loop:loop
    close c1;
END; $$
DELIMITER ;
#### Update Cursor



mandatory to make cursor   " for update "
DELIMITER $$
CREATE PROCEDURE cursor6( )
BEGIN
                               declare a int(4);
        declare b varchar(15);
        declare c int(4);
        declare d int(2);
        declare finished int(4) default 1 ;
        declare c1 cursor for select * from emp for update;
        declare continue handler for not found set finished = 0;
            open c1;
    cursor_c1_loop : loop
                                              fetch c1 into a,b,c,d;
                                              if finished = 0 then
                                                              leave cursor_c1_loop;
                                              end if;
                                              update emp set sal = sal + 1;
                insert into temp values(c,b);
    end loop cursor_c1_loop ;

    commit;    < --- locks are Automatically Released when you rollback or commit

    close c1;
END; $$  DELIMITER ;
```

PARAMETERS

There are 3 type of parameters in procedure

1. **IN (by default) (fastest in terms or processing speed)**

  - Read only (value of parameter cannot be modify inside the procedure)
  - you can not assign a value to the parameter inside the procedure
  - can pass a constant, variable, and expression also
  - call by value
  - Example :-

```
   delimiter //

   create procedure in_demo( in y int(4))

   begin

                    /* set y = 100 ; */ ----- > this will give you error

                    insert into temp values( y, ' inside in_demo');

   end; //

   delimiter ;

call in_demo(10)


or


set @x = 10;

call in_demo(@x);
```

2. **OUT (Most Secure)**

- write only (you can assign a value to the parameter inside the procedure but you cannot read from it)
- can pass variable only
- call by reference
- used on public network e. g. username, password, OTP, etc.
- Example :-

```
delimiter //

create procedure out_demo( out y int(4))

begin

set y = 100 ;

insert into temp values( y, ' inside in_demo');

end; //

delimiter ;

set @x = 10;

call in_demo(@x);
```

3. **INOUT (Most powerful) (best functionality)**

- Read and Write
- can pass variable only
- call by references
- Example :-

```
delimiter //

create procedure inout_demo( out y int(4))

begin

set y = y*y*y ;

insert into temp values( y, ' inside in_demo');

end; //

delimiter ;
```

Stored Function (v imp)

- stored objects .
- objects that are stored in the database .
- Routine that returns a value directly and compulsorily.
- stored in the database in the COMPILED FORMAT
- global functions.
- etc. same as stored procedures.
- can pass in parameter only.
- stored function can be called in select statement.
- stored function can be called in DML commands also.

There are two types of functions

- **Deterministic**
- **Not Deterministic**
  - For the same input parameter, if the function returns the same result, it is considered deterministic otherwise the function is not deterministic .
  - you have to decide whether a stored function is deterministic or not .
  - if you declare it incorrectly, the stored function may produce an unexpected result, or available optimization is not used which degrades the performance
  - Syntax :-

```
delimiter //

create function fun1()

returns int

deterministic

begin

return 10;
```

```
end; //

delimiter ;
```

- **Function with procedure**
  - unlike procedure , a function cannot be called by itself, because a function returns a value, and that value has to be stored somewhere
  - a part of some there you will have to equate the function with a variable, or it has to be part of some expression.
  - if the function returns boolean value then you can directly use the function name as a condition for IF statement.
  - a function is normally used as a validation, a function is normally used for checking purposes, a function normally returns a boolean TRUE or FALSE value, and accordingly some future processing is to be undertaken .

To show which all function are created :-

```
show function status;

or

show function status where db = 'sakshi';
```

Grant and Revoke with function

```
## Grant ##

grant execute on function fun1 to 'prachi';



## Revoke ##

revoke execute on function fun1 from 'prachi';
```

TRIGGERS

- stored objects.
- objects that are stored in the database.
- e. g. CREATE .... tables, indexes, views, procedures, functions e. g. CREATE ....
- triggers are present in some of the RDBMS.
- Routine (set of commands) that gets executed AUTOMATICALLY whenever some EVENT takes place.
- Triggers are written on tables.
- maintain logs (audit trails) of insertions
- all triggers are at server level (you can perform the DML operation using any front-end software, the triggers will always execute).
- In MySQL, all triggers are at Row level (will fire for each row).

- if the DML operation on table is failing then changes made by the trigger are automatically Rolled back.
- Your data will be consistent.
- within the trigger, you can have any processing (all MySQL-PL statements are allowed, e. g. loops, cursors, etc.).

EVENTS

Before Insert, After Insert

- used to Data cleansing ,automatic updating of related tables.
- you can used NEW variable in insert.
- example :-

```
delimiter //

create trigger trg1()

before insert

on emp for each row

begin

insert into temp values(1,'inserted');

set new.ename = upper(new.ename);

end; //

delimiter ;
```

Before Delete, After Delete

- used to implement Recycle bin in the event of delete.
- you can used OLD variable in delete.

```
delimiter //

create delete trg1()

before insert

on emp for each row

begin

        insert into temp values(1,'deleted');

        set new.ename = upper(old.sal,old.ename);

end; //

delimiter ;
```

Before Update, After Update

- inside the update triggers, you can use both OLD and NEW variable .

```
delimiter //

create delete trg1()

before update

on emp for each row

begin

        update deptot set saltot = saltot - old.sal where deptno = old.deptno;

        set new.ename = upper(old.sal,old.ename);

end; //

delimiter ;
```

There Are Two MySQL Created Variable For Trigger

NEW

- it is hold the new data .

OLD

- it is hold the old data.

- **maintain 2 or more copies of the table at the time of insert (automatic data duplication or mirroring) (for testing purposes)**.
- **inside the triggers, you cannot perform any DML operation on the same table (you will get an error of MUTATING TABLE)**

Show Triggers

```
show triggers;
```

Show Triggers from individual database

```
show triggers from [db_name];
```

Drop Triggers

- when you drop the table, the associated indexes and triggers are automatically drop, you cannot drop it manually.

Getting Ready for Normalisation :-

- Ask Client for sample data
- For a given transaction, make a list of fields.
- for all practical purposes, we can have a single table with all these column
- Give a suitable name to the suitable name to the table.
- remove computed columns
- Taking the user into confidence, strive for atomicity
- e. g. CADDR = AREA, STREET, PLOT_NO,BLDG_NAME, FLAT_NO, etc.
- with the help of user, make a list of filed properties (do this for every filed).
- get the user sign off.
- End of user interaction.
- finalize column datatype and widths.
- Finalize not null, unique and checked constraints.
- At this point data is in Un-Normalised Form (UNF).
- **UNF is a starting point of normalization**

onum	**no duplicate allow, numeric, no decimal, etc**
cut_num	
cut_name	
cut_mobile_no	
cut_address	**Street, Plot_no, Flat_no, building_name, Land_mark**
cut_city	
cut_pin_code	
order_date	**Todays date, date format e. g. DD/MM/YYYY etc.**
dely_date	
product_name	**alpha numeric, max width 50 chars**
product_code	
QTY	
RATE	

Repeating group :-

- Repeating group means those column these repeating every time .
- e. g **product_name** , **product_code**, **QTY** , **RATE**.
- **Key Elements :-**
- key elements means which elements those are have key constraints.
- **Non Key Elements :-**
- Non key elements means which elements those are not belongs to key constraints.

NORMALISATION (V. IMP)

- is a concept of table design.
- what table to create , what structure, column, datatypes, width, constraints;
- based on user requirement.
- is part of design phase.(1/6)
- Aim of Normalisation is to avoid the Data Redundancy (avoid unnecessary duplication of data) (this is a wastage of HD space).
- Aim is also to reduce the problem of insert, update, and delete.
- Normalisation is done from is input perspective.
- Normalisation is done from is forms perspective.
- VIEW THE ENTIRE APPLICATIONON A PER-TRANSACTION BASES AND THEN YOU NORMALISE EACH PER-TRANSACTION SEPARATELY.
- e. g. CUST_PLACE _AN_ORDER. CUST_MAKES_PAYMENT, CUST_CANCELS_THE_ORDER, GOOD_ARE_DELIVERED, etc.

Steps for Normalization :-

2. Remove the repeating group into a new table.
3. Key element will be PK of new table.
4. (This step may or may not be required) Add the primary key to original table to the new table to give you a composite PK.

First Normal Form

- Repeating groups are removed from table design.
- **1 : Many** Relationship is always encountered here.
- 25% table would in First-NF.

1. only the table with composite PK are examined.
2. Those non-key element that are not dependent on the entire composite PK, they are to be removed into a new table.
3. Key element on which originally dependent, it is to be added to the new table and it will be the PK of new table.

Second Normal Form

- Every column is Functionally dependent on the PK (known as Functional Dependency).
- Functional Dependency (without PK that column cannot function).
- 67% table would in SNF.

4. Only the non-key element are examined for inter-dependencies.
5. The inter-dependent column are to be removed into a new table.
6. key elements will be PK of new table; and the PK of new table it is to be added/retained in the original table for relationship purposes.

Third Normal Form (TNF)

- Transitive dependencies (inter-dependencies) are removed from table design
- 8% table would in TNF.
- what tables to create, what structure, column, datatype, widths, constraints.
- primary key is by product for normalization.

- it is remove the foreign key transitive dependencies between two table;

Forth Normal Form

- extension to 3rd Normal Form.
- also known as Boyce-Codd Normal Form (BCNF).
- you may or not implement 4th Normal Form.
- Normally used to public network.
- used to protect the accuracy of data.

De- Normalisation

- if the data is large , if the select are slow. you add an extra column to the table and store the data in it .
- used to improve the performance, to improve the speed of select statements.
- normally done for computed columns, expressions, formal column, summary column, function based column, etc.
- e. g. 1) item_total = qty*rate , 2) o_total = sum(item_total);
- in some situation you want to create an extra table altogether and store the totals over there.