Day 9

Stored Procedure

```
CREATE TABLE students
(
    roll_number INTEGER,
    name VARCHAR( 50 ),
    marks FLOAT,
    CONSTRAINT PK_STUDENTS PRIMARY KEY(roll_number)
);
```

```
INSERT INTO students
VALUES
(1,'Yogesh',85),
(2,'Rajiv',56),
(3,'Ketan',78),
(4,'Akash',62);
```

- Define stored procedure to insert record into student table

```
DELIMITER $$
CREATE PROCEDURE sp_insert_student(
IN roll_number INTEGER,
IN name VARCHAR(50),
IN marks FLOAT)
BEGIN
    INSERT INTO students
    VALUES
    (roll_number, name, marks);
END $$
DELIMITER ;
```

```
SHOW PROCEDURE STATUS;
SHOW PROCEDURE STATUS WHERE DB='dac_db';
SHOW PROCEDURE STATUS LIKE 'sp%';
```

```
CALL sp_insert_student(5,'Amit',74);
SELECT * from students;

SET @roll_number = 6;
SET @name = 'sandeep';
SET @marks = 45;
CALL sp_insert_student(@roll_number,@name,@marks);
SELECT * from students;
```

- Define stored procedure to update record into student table

```
DELIMITER $$
DROP PROCEDURE IF EXISTS sp_update_student;
CREATE PROCEDURE sp_update_student(
IN pRoll_number INTEGER,
IN pMarks FLOAT)
BEGIN
    UPDATE students
    SET marks = pMarks
    WHERE roll_number = pRoll_number;
END $$
DELIMITER ;
```

```
CALL sp_update_student( 4, 68 );
SELECT * FROM students;
```

- Define stored procedure to DELETE record from student table

```
DELIMITER $$
DROP PROCEDURE IF EXISTS sp_delete_student;
CREATE PROCEDURE sp_delete_student(
IN pRoll_number INTEGER )
BEGIN
    DELETE FROM students
    WHERE roll_number = pRoll_number;
END $$
DELIMITER ;
```

```
CALL sp_delete_student( 3 );
SELECT * FROM students;
```

Handler For Error Code

```
DELIMITER $$
CREATE PROCEDURE sp_insert_student(
IN roll_number INTEGER,
IN name VARCHAR(50),
IN marks FLOAT)
BEGIN
    -- DECLARE CONTINUE HANDLER FOR 1062 SELECT 'Duplicate entry';

    /* DECLARE CONTINUE HANDLER FOR 1062
    BEGIN
        SELECT 'Duplicate entry';
    END; */

    /* DECLARE DUPLICATE_ENTRY CONDITION FOR 1062;
    DECLARE CONTINUE HANDLER FOR DUPLICATE_ENTRY
    BEGIN
        SELECT 'Duplicate entry';
    END; */

    DECLARE status INTEGER DEFAULT 1;
    -- DECLARE CONTINUE HANDLER FOR 1062 SET status = 0;
    DECLARE CONTINUE HANDLER FOR 1062
    BEGIN
        SET status = 0;
    END;
    INSERT INTO students
    VALUES
    (roll_number, name, marks);
END $$
DELIMITER ;
```

Cursor

- Iteration / Traversing is process of visiting element in collection.
- Records retrurned by select query is called result set.
- If we want to visit rows of result set one by one then we should use cursor.
- Cursor is based of SELECT statement.
- SELECT STATEMENT can contain ORDER BY, LIMIT, GROUP BY, JOINS and SUB QUERIES.
- Cursor is a variable which is used to visit each row one by one.
- Syntax:

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

- Cursor declaration to access name of each row.

```
DECLARE cur CURSOR FOR SELECT name FROM students;
```

- Cursor declaration to access name and marks of each row.

```
DECLARE cur CURSOR FOR SELECT name, marks FROM students;
```

- Cursor declaration to access all values from each row.

```
DECLARE cur CURSOR FOR SELECT * FROM students;
```

- Syntax to fetch row

```
FETCH cursor_name INTO var_name ...;
```

Steps to use Cursor

1. Declare variables as per requirement.
2. Declare Cursor
3. Declare NOT FOUND handler
4. Open cursor
5. FETCH record into variable
6. Close Cursor

- Declare cursor to get name of all the students from student table.

```
DELIMITER $$
CREATE PROCEDURE sp_fetch_name( INOUT name_list VARCHAR(500) )
BEGIN
    DECLARE finished INTEGER DEFAULT 0;
    DECLARE sname VARCHAR(50) DEFAULT '';
    -- Declare cursor
    DECLARE cur CURSOR FOR SELECT name FROM students;
    -- Declare NOT FOUND Handler
    DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET finished = 1;
    -- Open Cursor
    OPEN cur;
    -- define labled loop
    label:LOOP
        FETCH cur INTO sname;
        IF finished = 1 THEN
            LEAVE label;
        END IF;
        SET name_list = CONCAT( sname,',',name_list);
    END LOOP label;
    -- Close Cursor
    CLOSE cur;
END $$
DELIMITER ;
```

```
SET @list = '';
CALL sp_fetch_name( @list );
SELECT @list FROM DUAL;
```

- "OPEN cursor_name" statement execute SELECT statement and it attaches cursor to it(Result Set).
- "FETCH" statement fetches the next row for the SELECT statement associated with the specified cursor and advances the cursor pointer.
- If a row exists, the fetched columns are stored in the named variables.
- The number of columns retrieved by the SELECT statement must match the number of output variables specified in the FETCH statement.
- If no more rows are available, a No Data condition occurs with SQLSTATE value '02000'. To detect this condition, you can set up a handler for it (or for a NOT FOUND condition)
- "CLOSE cursor_name" statement closes a previously opened cursor.
- If not closed explicitly, a cursor is closed at the end of the BEGIN... END block in which it was declared.
- We can define cursor inside PL BLOCK(BEGIN...END).

```
DELIMITER $$
DROP PROCEDURE IF EXISTS sp_fetch_name;
CREATE PROCEDURE sp_fetch_name( INOUT name_list VARCHAR(500) )
BEGIN
    DECLARE finished INTEGER DEFAULT 0;
    DECLARE sname VARCHAR(50) DEFAULT '';
    -- Declare cursor
    DECLARE cur CURSOR FOR SELECT name FROM students;
    -- Declare NOT FOUND Handler
    DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET finished = 1;
    -- Open Cursor
    OPEN cur;
    -- define labled loop
    label:LOOP
        FETCH cur INTO sname;
        IF finished = 1 THEN
            LEAVE label;
        END IF;
        SET name_list = CONCAT( sname,',',name_list);
    END LOOP label;
    -- Close Cursor
    CLOSE cur;
    -- define another cursor

    SET finished = 0;
    -- Open Cursor
    OPEN cur;
    -- define labled loop
    label:LOOP
        FETCH cur INTO sname;
        IF finished = 1 THEN
            LEAVE label;
        END IF;
        SET name_list = CONCAT( sname,',',name_list);
    END LOOP label;
    -- Close Cursor
    CLOSE cur;

END $$
DELIMITER ;
```

```
SET @list = '';
CALL sp_fetch_name( @list );
SELECT @list FROM DUAL;
```

Characteristics of cursor

1. It is forward only. we can skip row or we can not move back.
2. It is read only.
3. MySQL cursors are asensitive cursors. i.e during traversing, if another user make chanages into the table then cursor returns updated data.

- During traversing if we dont want to update the record then we shoud use FOR UPDATE in SELECT query.

Stored Function

- It is server side program which is used to achive reusability.
- If predefined/library defined functions are insufficient to execute business logic then we should define function.
- Syntax:

```
CREATE FUNCTION sp_name (params,... )
RETURNS type [NOT] DETERMINISTIC
BEGIN
    routine_body
END
```

- We can not call stored function explicitly using CALL statement. To invoke a stored function, refer to it in an expression.
- Specifying a parameter as IN, OUT, or INOUT is valid only for a PROCEDURE. For a FUNCTION, parameters are always regarded as IN parameters.

```
DELIMITER $$
CREATE FUNCTION TOUPPERCASE( str VARCHAR(50)) RETURNS VARCHAR( 50 )
DETERMINISTIC
```

```
BEGIN
    DECLARE temp VARCHAR(50) DEFAULT '';
    SET temp = CONCAT('Hello,',' ', UPPER(str));
    RETURN temp;
END $$
DELIMITER ;
```

```
-- CALL TOUPPERCASE('dac'); -- Error
SELECT TOUPPERCASE('dac') "Name" FROM DUAL;
```

- Function can not return multiple values.
- Types of Function:

1. DETERMINISTIC Function:
    - For same input parameters, if function generates same result then it is called DETERMINISTIC function.
    - BIN(10);
    - 90% functions are DETERMINISTIC
2. NON DETERMINISTIC Function:
    - For same input parameters, if function generates different result then it is called non DETERMINISTIC function.
    - e g: count_exp(joinDate);

Trigger

- A trigger is a named database object that is associated with a table, and that activates when a particular event occurs for the table.
- Syntax:

```
CREATE TRIGGER trigger_name
trigger_time trigger_event
ON tbl_name FOR EACH ROW [trigger_order]
BEGIN
    trigger_body
END
```

- trigger_time:{ BEFORE | AFTER }
- trigger_event:{ INSERT | UPDATE | DELETE }
- trigger_order: { FOLLOWS | PRECEDES } other_trigger_name
- The trigger becomes associated with the table named tbl_name, which must refer to a permanent table. You cannot associate a trigger with a TEMPORARY table or a view.
- Following trigger_event values are permitted:

1. INSERT: The trigger activates whenever a new row is inserted into the table.

2. UPDATE: The trigger activates whenever a row is modified.

3. DELETE: The trigger activates whenever a row is deleted from the table.

- TCL operations are not permitted in trigger.
- It doesnt take any parameter or return any value. NEW and OLD are implicit variables available for trigger.
- If we want to maintain log of DML operations then we should use trigger.

```
CREATE TABLE accounts
(
    number INTEGER,
    name VARCHAR(50),
    balance FLOAT,
    type VARCHAR(50)
);
CREATE TABLE transactions
(
    number INTEGER,
    balance FLOAT,
    type VARCHAR( 50 )
);
INSERT INTO accounts
VALUES
(101,'Amol',85000,'Saving'),
(102,'Rupesh',30000,'Current'),
(103,'Sandeep',185000,'Loan');
```

```
DELIMITER $$
CREATE TRIGGER tgr_insert AFTER INSERT ON accounts FOR EACH ROW
BEGIN
    INSERT INTO transactions
    VALUES( NEW.number, NEW.balance, NEW.type );
END $$
DELIMITER ;
```

```
--SHOW TRIGGERS [FROM db_name] [like_or_where;
SHOW TRIGGERS FROM dac_db;
```

```
DELIMITER $$
CREATE TRIGGER tgr_update AFTER UPDATE ON accounts FOR EACH ROW
BEGIN
    INSERT INTO transactions
    VALUES( OLD.number, OLD.balance, OLD.type );

    INSERT INTO transactions
    VALUES( NEW.number, NEW.balance, NEW.type );
END $$
DELIMITER ;
```