BLG 317E
DATABASE SYSTEMS

WEEK 7

Structured Query Language (SQL)
Stored Procedures and Triggers

Topics

- Variables
- Built-in Functions
- Stored Functions
- Stored Procedures
- Procedural Language statements
- Random Data Generation
- Cursors
- Triggers

Procedural Language Extensions to Standard SQL

- Standard SQL (ANSI/ISO SQL) is a declarative language.
- Standard SQL does not have any procedural commands such as IFs, loops, stored functions, stored procedures, etc.
- However, most relational DBMSs have their own procedural language extensions.
- Their procedural language syntaxes are different than the others, and they are not compatible.

Procedural Languages

Relational DBMS	Procedural Language
Oracle	**PL/SQL** (Procedural Language / SQL)
Microsoft SQL Server	**T-SQL** (Transact-SQL)
MySQL	**SQL/PSM** (SQL / Persistent Stored Modules)
PostgreSQL	**PL/pgSQL**
DB2	**SQL PL**

Variables in MySQL

- MySQL supports defining and using variables.
- A variable can contain only simple values (scalars, not arrays).
- Variable name should start with the @ symbol.
- The value can be assigned directly, or can be assinged with the result of another SELECT command.
- Once a variable has been defined, it can be used in any SQL command, until the connection session to the MySQL server finishes.
- There are two basic methods of defining and assigning values to a variable.

  SET command

  SELECT command

Example: SALES Table

Suppose the following example table is in the database.

SALES table

sale_id	product_name	price	quantity	month
101	Orange	3.0	22	Jan
102	Kiwi	0.8	34	Jan
103	Orange	1.5	23	Feb
104	Apple	2.0	38	Feb
105	Cherry	2.7	44	Feb
106	Apricot	1.2	78	Feb
107	Orange	3.0	76	March
108	Cherry	4.1	60	March
109	Apple	1.6	48	March

Query1 : Retrieve a list of all records in SALES table.

Each record in the result-set should also show the average price information. Sort the output by product name.

Query

```
WITH RESULT AS (
  SELECT AVG(Price) AS average
  FROM SALES
)
SELECT *
FROM SALES, RESULT
order by Product_Name;
```

Common Table Expression is used as subquery. (Using the WITH clause)

Result-set

sale_id	product_name	price	quantity	month	average
104	Apple	2.0	38	Feb	2.21111
109	Apple	1.6	48	March	2.21111
106	Apricot	1.2	78	Feb	2.21111
105	Cherry	2.7	44	Feb	2.21111
108	Cherry	4.1	60	March	2.21111
102	Kiwi	0.8	34	Jan	2.21111
101	Orange	3.0	22	Jan	2.21111
103	Orange	1.5	23	Feb	2.21111
107	Orange	3.0	76	March	2.21111

Query2 :

Step1) Calculate the average of all prices in SALES table, and save the calculated average into a variable.

Step2) Update all records in SALES table whose price field are less than the averge price, so that their new prices are assigned with the average price.

Step3) Get a list of all records from SALES table to see the updated new prices.

Step1)

There are different ways to define and assign **average** variable. You may choose any of the followings.

```
set  @average  = ( select  AVG(Price)
                   from SALES );

Query OK, 0 rows affected (0.35 sec)
```

```
select  @average :=  AVG(Price)
                     from SALES;
```

```
select  AVG(Price)
from SALES  into @average;
```

To display the value of average variable, a SELECT command can be used.

```
mysql> select  @average;

+-------------+
| @average    |
+-------------+
| 2.211111111 |
+-------------+

1 row in set (0.00 sec)
```

Step2)

A separate **UPDATE** command is written, which uses the value of **average** variable that was calculated in Step1.

```
update SALES
set price = @average
where price < @average;

Query OK, 5 rows affected, 5 warnings (0.39 sec)
Rows matched: 5  Changed: 5  Warnings: 5
```

Step3)

The UPDATE command does not return any result-set.

To see the results of UPDATE command, a separate SELECT command should be entered.

```
select * from SALES
order by product_name;
```

sale_id	product_name	price	quantity	month
104	Apple	2.2	38	Feb
109	Apple	2.2	48	March
106	Apricot	2.2	78	Feb
105	Cherry	2.7	44	Feb
108	Cherry	4.1	60	March
102	Kiwi	2.2	34	Jan
101	Orange	3.0	22	Jan
103	Orange	2.2	23	Feb
107	Orange	3.0	76	March

Topics

- Variables
- Built-in Functions
- Stored Functions
- Stored Procedures
- Procedural Language statements
- Random Data Generation
- Cursors
- Triggers

SQL Built-in Functions

Category	Built-In Functions
String Functions	ASCII CHAR_LENGTH CHARACTER_LENGTH CONCAT CONCAT_WS FIELD FIND_IN_SET FORMAT INSTR LCASE LEFT LENGTH LOCATE LOWER LPAD LTRIM MID POSITION REPEAT REPLACE REVERSE RIGHT RPAD RTRIM SPACE STRCMP SUBSTR SUBSTRING SUBSTRING_INDEX TRIM UCASE UPPER
Numeric Functions	ABS ACOS ASIN ATAN ATAN2 AVG CEIL CEILING COS COT COUNT DEGREES DIV EXP FLOOR GREATEST LEAST LN LOG LOG10 LOG2 MAX MIN MOD PI POW POWER RADIANS RAND ROUND SIGN SIN SQRT SUM TAN TRUNCATE

SQL Built-in Functions

Category	Built-In Functions
Date Functions	ADDDATE ADDTIME CURDATE CURRENT_DATE CURRENT_TIME CURRENT_TIMESTAMP CURTIME DATE DATEDIFF DATE_ADD DATE_FORMAT DATE_SUB DAY DAYNAME DAYOFMONTH DAYOFWEEK DAYOFYEAR EXTRACT FROM_DAYS HOUR LAST_DAY LOCALTIME LOCALTIMESTAMP MAKEDATE MAKETIME MICROSECOND MINUTE MONTH MONTHNAME NOW PERIOD_ADD PERIOD_DIFF QUARTER SECOND SEC_TO_TIME STR_TO_DATE SUBDATE SUBTIME SYSDATE TIME TIME_FORMAT TIME_TO_SEC TIMEDIFF TIMESTAMP TO_DAYS WEEK WEEKDAY WEEKOFYEAR YEAR YEARWEEK
Other Functions	BIN BINARY CASE CAST COALESCE CONNECTION_ID CONV CONVERT CURRENT_USER DATABASE IF IFNULL ISNULL LAST_INSERT_ID NULLIF SESSION_USER SYSTEM_USER USER VERSION

Exercise: SQL Numeric Functions for calculations on Table columns

- Suppose the **TRIANGLES table** contains triangle ID numbers, and three edge lengths (**A, B, C**) of several triangles.
- Write SQL statement to create the table.
- Write SQL statements to add all data given below to the table.
- **Query:** Write a **SELECT** statement to calculate and display the **Perimeter** and the **Area** values of all triangles.

TRIANGLES table

Triangle ID	A	B	C
1	30	40	50
2	10	10	10
3	40	90	100
4	50	40	20

Calculation Formulas

$$Perimeter = a + b + c$$

$$S = (a + b + c)/2$$

$$Area = \sqrt{S(S-a)(S-b)(S-c)}$$

SOLUTION

```
create table TRIANGLES (idnum int unique,
                        A int, B int, C int);
```

```
insert into TRIANGLES values
  (1, 30, 40, 50),
  (2, 10, 10, 10),
  (3, 40, 90, 100),
  (4, 50, 40, 20);
```

Query

```
select *,
       A+B+C  AS  Perimeter,
       ROUND ( SQRT( (A+B+C)/2 *
                     ( (A+B+C)/2 - A) *
                     ( (A+B+C)/2 - B) *
                     ( (A+B+C)/2 - C) )
             )  AS  Area
from TRIANGLES;
```

Result of SELECT Query

idnum	A	B	C	Perimeter	Area
1	30	40	50	120	600
2	10	10	10	30	43
3	40	90	100	230	1798
4	50	40	20	110	380

Topics

- Variables
- Built-in Functions
- Stored Functions
- Stored Procedures
- Procedural Language statements
- Random Data Generation
- Cursors
- Triggers

Stored Function Syntax in MySQL

- A stored function is a stored prstudentam that returns a single value.
- It is mostly used to calculate mathematical formulas or business rules that are reusable by SQL statements or stored procedures.
- Different from a stored procedure, you can use a stored function in SQL statements wherever an expression is used.

Syntax

```
DELIMITER $$

CREATE FUNCTION function_name(parameter_list)
RETURNS datatype
BEGIN
  -- Statements
END $$

DELIMITER ;
```

Stored Functions in MySQL

- By default, all parameters are input only.
- Specify the data type of the return value in the RETURNS statement, which can be any valid SQL data types.
- Write the code in the body of the stored function in the BEGIN-END block.
- Inside the BEGIN-END block, you need to specify at least one RETURN statement.
- The RETURN statement returns a value to the calling prstudentams.
- Whenever the RETURN statement is reached, the execution of the stored function is terminated immediately.
- A stored function becomes the part of database, like the built-in SQL function.
- Any SQL command (SELECT,INSERT,UPDATE,DELETE) can call a stored function.

Example1: Stored Function (average_func)

- The stored function below takes two integer numbers as parameters.
- It calculates and returns the average as a float number.

```
DELIMITER $$
CREATE FUNCTION average_func(x int, y int)
RETURNS float
DETERMINISTIC
BEGIN
    DECLARE avg float;
    SET avg = (x+y)/2;
    RETURN (avg);
END$$
DELIMITER ;
```

- The stored function average_func can be directly called from MySQL command-line prompt.
- Function takes parameters, and returns the result value.

Query

```
select  average_func(10, 17);
```

Result value

```
+-----------------------+
| average_func(10, 17)  |
+-----------------------+
|                  13.5 |
+-----------------------+
```

Example Table

- Suppose the following table is in the database.
- Write a query to calculate and display the average of two grades for each student.
- Query should call the stored function average_func.

GRADES table

student	grade1	grade2
Reynolds	73	84
Spalding	100	90
Hughart	54	61
Lewis	43	84

Query

```
select *,
       average_func(grade1, grade2)
       as avrg

from GRADES;
```

Result-set of query

student	grade1	grade2	avrg
Reynolds	73	84	78.5
Spalding	100	90	95
Hughart	54	61	57.5
Lewis	43	84	63.5

Example2: Stored Function (class_average_func)

- This stored function takes sütun (column code) as a parameter.
- If sütun is 1, then function calculates the class average of grade1 values in GRADES table. Otherwise, it calculates average of grade2.
- Function returns a float number.

```
DELIMITER $$
CREATE FUNCTION class_average_func(column int)
RETURNS float
DETERMINISTIC
BEGIN
    DECLARE avrg float;
    IF column == 1 THEN
       set avrg = (select AVG(grade1) from GRADES );
    ELSE
       set avrg = (select AVG(grade2) from GRADES );
    END IF;
    RETURN (avrg);
END$$
DELIMITER ;
```

- In the following example, the stored function is called twice in the same SELECT statement, with different parameters.
- Two averages are calculated and returned (for grade1 and grade2)

Query

```
select class_average_func(1) as avg1,
       class_average_func(2) as avg2;
```

Result values

```
+------+-------+
| avg1 | avg2  |
+------+-------+
| 67.5 | 79.75 |
+------+-------+
```

- In order to calculate the general average of grades for entire class, the following three queries can be used in the given order.

Queries

```
select class_average_func(1)
       into @avg1;

select class_average_func(2)
       into @avg2;

select  @avg1, @avg2,
        (@avg1 + @avg2) /2
        as general_average;
```

Result values

```
+-------+-------+-----------------+
| @avg1 | @avg2 | general_average |
+-------+-------+-----------------+
|  67.5 | 79.75 |              74 |
+-------+-------+-----------------+
```

Topics

- Variables
- Built-in Functions
- Stored Functions
- Stored Procedures
- Procedural Language statements
- Random Data Generation
- Cursors
- Triggers

SQL Script Files

- An SQL script file can contain SQL statements (CREATE, DROP, ALTER, SELECT, INSERT, DELETE, UPDATE, etc.)
- Usually, frequently used queries are saved in **external script files**.
- A user can manually load the external SQL script file into MySQL command-line or, to Workbench editor, then execute it.
- Instead of saving SQL queries to an external script file, we can also store them directly in the database as **Stored Procedures.**
- Stored Procedures are an alternative method of saving frequently used SQL queries.

Stored Procedures

- A stored procedure is a set of SQL statements that can be executed on the database.
- It is stored as an object in the database.
- A stored procedure allows for code that is run many times.
- A stored procedure is similar to a stored function.
- In addition to SQL statements, it can also contain Procedural Language commands such as IFs and loops.
- The difference is that a stored function is called by a SELECT command, but a stored procedure is called by a CALL statement.
- A CALL command must be used to call a stored procedure.
- Complex queries can be written as stored procedures.
- Instead of writing ad-hoc (un-planned) queries every time a request is made, a stored procedure helps save time.

Advantages of Stored Procedures

- The first time a stored procedure is invoked, MySQL looks up for the name in the database catalog, compiles the stored procedure's code, place it in a memory area known as a cache, and execute the stored procedure.

- If the same stored procedure is invoked in the same session again, MySQL just executes the stored procedure from the cache without having to recompile it.

- A stored procedure may contain control flow statements (Procedural Language commands) such as IF, CASE, and LOOP that allow to implement the code in the procedural way.

- A stored procedure can call other stored procedures or stored functions.

Stored Procedure Syntax in MySQL

- In the Statements part below, any SQL statements such as SELECT, INSERT, DELETE, UPDATE can be used.
- The parameters are optional.
- There can not be a RETURN command in a stored procedure.
- Procedure can return a result-set by using a SELECT command in the statements part.

Syntax

```
DELIMITER $$

CREATE PROCEDURE stored_proc_name(parameters)
BEGIN
  -- Statements
END $$

DELIMITER ;
```

Parameters in Stored Procedure

- A parameter in a stored procedure has one of three modes: IN, OUT, or INOUT.
- IN is the default mode. The calling prstudentam has to pass an argument to the stored procedure.
- The value of an OUT parameter is assigned inside the stored procedure and its value is passed to the calling prstudentam.
- An INOUT parameter is a combination of IN and OUT parameters. The calling prstudentam may pass the argument, and the stored procedure can modify the INOUT parameter, and pass the new value back to the calling prstudentam.

Syntax

```
[IN | OUT | INOUT]  parameter_name  datatype
```

Example1: Stored Procedure (compute_SP)

- The following stored procedure takes 4 parameters.
- The x and y parameters are used as inputs.
- The sum and multiplication parameters are used as outputs.
- There is no RETURN command.
- The calculated result values are returned via the output parameters.

```
DELIMITER $$
CREATE PROCEDURE compute_SP(IN x int, IN y int,
                            OUT summation int,
                            OUT multiplication int)
BEGIN
  SET summation = x+y;
  SET multiplication = x*y;
END$$
DELIMITER ;
```

- Calling the compute_SP stored procedure from command-line.

Queries

```
call compute_SP(4, 5, @result1, @result2);
```

```
select 4 num1,
       5 num2,
       @result1 as summation,
       @result2 as product;
```

Result values

```
+-------+-------+---------+--------+
| num1  | num2  |summation|product |
+-------+-------+---------+--------+
|     4 |     5 |       9 |     20 |
+-------+-------+---------+--------+
```

Example2: Stored Procedure (all_avgs_SP)

- The following stored procedure takes no input/output parameters.
- There is no RETURN command.
- The SELECT statement inside the procedure returns a result-set.

```
DELIMITER $$
CREATE PROCEDURE all_avgs_SP()
BEGIN

    select *, average_func(grade1, grade2) as avrg
    from GRADES;

END$$
DELIMITER ;
```

- The stored procedure is called from command line.
- It returns a result-set, containing grades averages for each student.

Query

```
CALL  all_avgs_SP();
```

Result-set

```
+----------+------+------+------+
| student  |grade1|grade2| avrg |
+----------+------+------+------+
| Reynolds |   73 |   84 | 78.5 |
| Spalding |  100 |   90 |   95 |
| Hughart  |   54 |   61 | 57.5 |
| Lewis    |   43 |   84 | 63.5 |
+----------+------+------+------+
```

Dropping Stored Functions and Procedures from Database

- The following drop commands deletes the specified stored functions and the stored procedure from the database.

```
DROP FUNCTION average_func;

DROP FUNCTION class_average_func;
```

```
DROP PROCEDURE compute_SP;

DROP PROCEDURE all_avgs_SP;
```

Example Tables

- Suppose the following is the schema definition of two related tables.
- CategoryID is a primary key in CATEGORIES table, and a foreign key in PRODUCTS table.

```
CATEGORIES (CategoryID, CategoryName)

PRODUCTS (ProductID, ProductName, CategoryID)
```

CATEGORIES table

CategoryID	CategoryName
1	Beverages
2	Condiments
3	Confections
4	Dairy Products
5	Frozen Foods
6	Grains/Cereals
7	Meat/Poultry
8	Produce
9	Seafood
10	Vegetables

PRODUCTS table

ProductID	ProductName	CategoryID	Price
1	Berry Spread	2	25
2	Boston Crab	8	18
3	Cajun Seasoning	2	22
4	Chais Coffee	1	18
5	Chang Syrup	1	19
6	Chartreuse verte	1	18
7	Chocolade	3	13
8	Chocolate Biscuits	3	9
9	Clam Chowder	8	10
10	Cote de Blaye	1	264
11	Cranberry Sauce	2	40
12	Dried Tomatoes	7	30
....	....	....	....

Example3: Stored Procedure (statistics_SP)

- The following stored procedure takes the category_name as input parameter.

- If **category_name**is NULL, then procedure performs **LEFT JOIN** operation from **CATEGORIES** table to **PRODUCTS** table.
- It calculates **how many products** are there in each category, and the **average price** of that category.

- If **category_name**is not NULL, then procedure performs **Implicit Inner Join** operation.
- It calculates statsictics only for the specified category name.
- The **LIKE operator** is used for partial matching between the parameter and CategoryName field in CATEGORIES table.

Part1

```
DELIMITER $$
CREATE PROCEDURE statistics_SP(IN   category_name
                                    varchar(15) )
BEGIN

    IF category_nameIS NULL THEN

       select CategoryName,
              IFNULL(COUNT(Price), 0) as ProdCount,
              IFNULL(AVG(Price), 0)   as ProdAvgPrice
       from CATEGORIES
       LEFT JOIN PRODUCTS
       ON CATEGORIES.categoryID = PRODUCTS.categoryID
       group by CategoryName
       order by CategoryName;
```

Part2

```
    ELSE

       select CategoryName,
              IFNULL(COUNT(Price), 0) as ProdCount,
              IFNULL(AVG(Price), 0) as ProdAvgPrice
       from CATEGORIES, PRODUCTS
       where CATEGORIES.categoryID = PRODUCTS.categoryID AND
             CategoryName LIKE CONCAT('%', category_name, '%')
       group by CategoryName
       order by CategoryName;

    END IF;

END$$
DELIMITER ;
```

Query1 : Get a list of **all categories** and calculate their statistics, regardless of the category name.

```
call  statistics_SP(null);
```

Result-set

CategoryName	ProdCount	ProdAvgPrice
Beverages	8	50.2500
Condiments	7	21.8571
Confections	11	24.0909
Dairy Products	5	21.4000
Frozen Foods	3	14.6667
Grains/Cereals	3	86.6667
Meat/Poultry	3	33.0000
Produce	10	22.1000
Seafood	0	0.0000
Vegetables	0	0.0000

Query2 : Get only a list of categories and their statistics, whose category name contains the **'Con'** word.

```
call  statistics_SP('Con');
```

Result-set

CategoryName	ProdCount	ProdAvgPrice
Condiments	7	21.8571
Confections	11	24.0909

Topics

- Variables
- Built-in Functions
- Stored Functions
- Stored Procedures
- Procedural Language statements
- Random Data Generation
- Cursors
- Triggers

IF Statement Syntax

- The ELSE clause is optional.
- Statements can be any SQL commands, or Procedural Language commands.
- Nested IFs and loops can be used.

Syntax

```
IF condition THEN
   Statements1;
ELSE
   Statements2;
END IF;
```

LOOP Statement Syntax

- The LOOP coomand works as an endless loop.
- An IF statement and LEAVE command is necessary to exit from the endless loop.
- The label is optional.

Syntax

```
[label]: LOOP

    Statements;

  -- Terminate the loop
    IF condition THEN
        LEAVE [label];
    END IF;

    Statements;

END LOOP;
```

Example: LOOP Statement

```
DELIMITER $$
CREATE PROCEDURE Loop1_SP()
BEGIN
    DECLARE counter       INT  DEFAULT  10;
    DECLARE result_str    VARCHAR(200) DEFAULT '';

    KeepOn: LOOP

       SET result_str = CONCAT(result_str, counter, ' , ');
       SET counter = counter + 10;

       IF counter > 100 THEN
          LEAVE KeepOn;
       END IF;

    END LOOP;
   -- Display the result
    SELECT result_str;
END$$
DELIMITER ;
```

Query

```
call  Loop1_SP(null);
```

Result string

```
+-----------------------------------------------------+
| result_str                                          |
+-----------------------------------------------------+
| 10 , 20 , 30 , 40 , 50 , 60 , 70 , 80 , 90 , 100 ,  |
+-----------------------------------------------------+

1 row in set (0.00 sec)
```

WHILE Statement Syntax

- The condition is checked first.
- If condition is True, then loop continues.

Syntax

```
[label:]
WHILE condition DO

   Statements

END WHILE [label]
```

Example: WHILE Statement

```
DELIMITER $$

CREATE PROCEDURE Loop2_func()
BEGIN
    DECLARE counter INT DEFAULT 10;
    DECLARE result_str   VARCHAR(200) DEFAULT '';

    WHILE counter <= 100 DO

       SET result_str = CONCAT(result_str, counter, ' , ');
       SET counter = counter + 10;

    END WHILE;

  -- Display the result
    SELECT result_str  ;

END$$
DELIMITER ;
```

REPEAT Statement Syntax

- It is similar to WHILE statement.
- First the Statements are executed once.
- Then, condition is checked.
- If condition is True, then loop continues again.

Syntax

```
[label:] REPEAT

    Statements

UNTIL condition

END REPEAT [label]
```

Example: REPEAT Statement

```
DELIMITER $$
CREATE PROCEDURE Loop3_SP()
BEGIN
    DECLARE counter      INT DEFAULT 10;
    DECLARE result_str   VARCHAR(200) DEFAULT ‘’;

    REPEAT
        SET result_str = CONCAT(result_str, counter, ' , ');
        SET counter = counter + 10;
    UNTIL counter > 100
    END REPEAT;

  -- Display the result
    SELECT result_str;

END$$
DELIMITER ;
```

Topics

- Variables
- Built-in Functions
- Stored Functions
- Stored Procedures
- Procedural Language statements
- Random Data Generation
- Cursors
- Triggers

Populating a Table with Random Data

- An SQL script or a Stored Procedures can be written to populate (adding dummy data) a table with random data.
- The table with random that can be used for testing purposes.
- In this example, a stored function (generate_rand_word) and a stored procedure (add_random_records) will be written to add random data records to a table.
- Firstly, the following CREATE TABLE command is executed.

```
CREATE TABLE PRODUCTS(
    product_id INTEGER
               PRIMARY KEY AUTO_INCREMENT,
    product_name VARCHAR(20),
    category_id INTEGER,
    price numeric(3, 1)
);
```

Stored Function (generate_rand_word)

- The following stored function takes the maximum string length (length) as input parameter.
- Function generates and returns a random string with all upper-case characters.
- It calls the following built-in SQL functions.

RAND() : Returns a random float number between 0-1.

LENGTH() : Returns number of characters in a string.

FLOOR() : Rounds downward as float number.

SUBSTRING() : Extracts a sub-string from an existing string.

CONCAT() : Concatanes second string at the end of first string.

Stored Function (generate_rand_word)

- The function takes the maximum string length (length) as input parameter.
- It generates and returns a string with all upper-letter characters.

```
DELIMITER $$
CREATE FUNCTION generate_rand_word(length int)
RETURNS varchar(20)
DETERMINISTIC
BEGIN
    SET @word  = '';
    SET @letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
    SET @array_length = LENGTH(@letters);
    SET @i = 0;

    WHILE (@i < length) DO
        set @position  = FLOOR(RAND() * @array_length) + 1;
        set @single_letter = SUBSTRING(@letters, @position, 1);
        SET @word   = CONCAT(@word, @single_letter);
        SET @i = @i + 1;
    END WHILE;

    RETURN @word;
END$$
DELIMITER ;
```

The stored function can be tested, by directly calling from MySQL command-line, or from MySQL Workbench query editor.

Query1

```
select generate_rand_word(10) as word;
```

Result

```
+------------+
| word       |
+------------+
| USHGIXNVRX |
+------------+
```

Query2

```
select generate_rand_word(20) as word;
```

Result

```
+----------------------+
| word                 |
+----------------------+
| XHPCUQSTQXNALDJKZQEB |
+----------------------+
```

Stored Procedure

- The following stored procedure takes the maximum number of rows (records) (row_count) as input parameter.
- In a loop, procedure performs followings:
  - The product_id field is an auto-increment primary key, which is assigned automatically as sequential number by DBMS.
  - For product_name field, the stored function (generate_rand_word) is called.
  - It calls the following built-in SQL functions.

    MOD() : Returns remainder of divison of two numbers

    ROUND() : Rounds upward a float number.

    RAND() : Returns a random float number between 0-1.
  - For category_id, and price fields the built-in SQL function RAND() is called.
  - Procedure uses NSERT command to add a new record to PRODUCTS table.

Stored Procedure (add_random_records)

```
DELIMITER $$
CREATE PROCEDURE add_random_records(IN row_count int)
BEGIN
  DECLARE i INT DEFAULT 0;
  set @N = 5;

  IF row_count IS NOT NULL THEN
     set @N = row_count;
  END IF;

  WHILE i < @N DO
    set @word    = generate_rand_word(20);
    set @category = mod(round(rand()*10), 6) + 1;
    set @price    = rand()*100;

    INSERT INTO PRODUCTS(product_name, category_id, price)
        VALUES (@word, @category, @price);

    SET i = i + 1;
  END WHILE;
END$$
DELIMITER ;
```

Calling the stored procedure:

Query1

```
mysql> call add_random_records(6);

Query OK, 1 row affected (0.66 sec)
```

Query2

```
select * from PRODUCTS;
```

Result-set

```
+------------+----------------------+-------------+-------+
| product_id | product_name         | category_id | price |
+------------+----------------------+-------------+-------+
|          1 | MSCHGHTWDDDJKVAPYZBH |           3 |  33.0 |
|          2 | YUDGVKPRQCQXQJBBDNFK |           6 |  29.5 |
|          3 | YVIFZJVCZQGIVHVHXTYM |           6 |  33.0 |
|          4 | ABEUIJVZNOEIBHHOWRWF |           5 |  55.1 |
|          5 | MSCILCGZFAMHAEVRWHUC |           2 |  29.8 |
|          6 | DVTESCFUGZEZGIYUAVZN |           6 |  22.6 |
+------------+----------------------+-------------+-------+
6 rows in set (0.00 sec)
```

Topics

- Variables
- Built-in Functions
- Stored Functions
- Stored Procedures
- Procedural Language statements
- Random Data Generation
- Cursors
- Triggers

Cursors

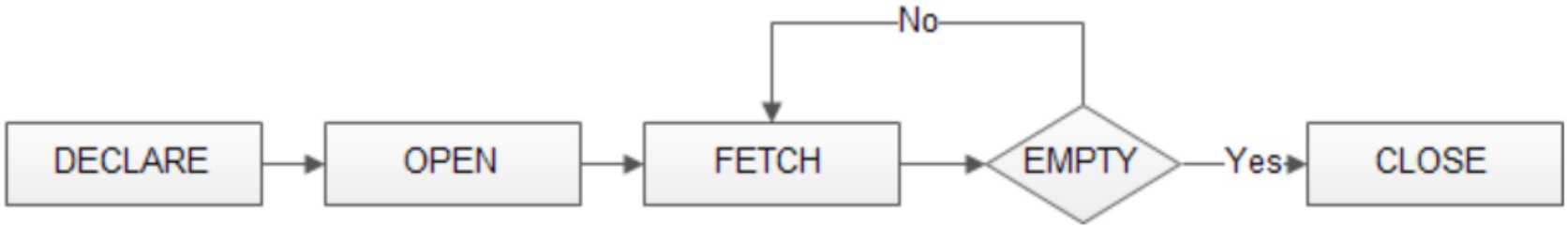
- A cursor is a pointer to a particular row (record) in a result-set of a SELECT statement.
- Standard SQL commands are set-based operations.
  - They work on a block of data.
  - Mostly set-based operations are preferred, due to speed.
- A cursor is used to retrieve data from database in a row-based operation.
  - A cursor allows to iterate rows (looping) returned by a query, and process each row individually, one row at a time.
  - Usage of cursors are preferred less, due to slowness.
- Cursors are written inside a stored procedure or stored function.

Cursors

- Cursors are part of Procedural Language extensions of SQL.
- Different DBMSs have different syntaxes for cursors.
- MySQL cursor limitations:
  - Read-only (can not be updated or deleted)
  - Not-scrolled (only sequential fetch allowed, can not fetch previous rows)
- Main steps of cursor usage:
  - Declare the cursor
  - Open the cursor
  - Fetch tuples one by one (in a loop)
  - Close the cursor

Steps for Using a Cursor

- The diagram below illustrates the steps to define and use a cursor.
- A cursor should be accessed thru a looping statement, and should always associate with a SELECT statement.



Stored Procedure Template for using a Cursor

```
DELIMITER $$
CREATE PROCEDURE stored_procedure_name(parameters )
BEGIN

     DECLARE variables_list;

     -- declare cursor for result-set
1    DECLARE cursor_name
          CURSOR FOR SELECT_statement;

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

3    OPEN cursor_name;
     KeepOn: LOOP
4         FETCH cursor_name INTO variables_list;
          IF finished_flag = 1 THEN
5              LEAVE KeepOn;
          END IF;

          Statements;

     END LOOP KeepOn;
6    CLOSE cursor_name;
END$$
DELIMITER ;
```

Steps for Using a Cursor

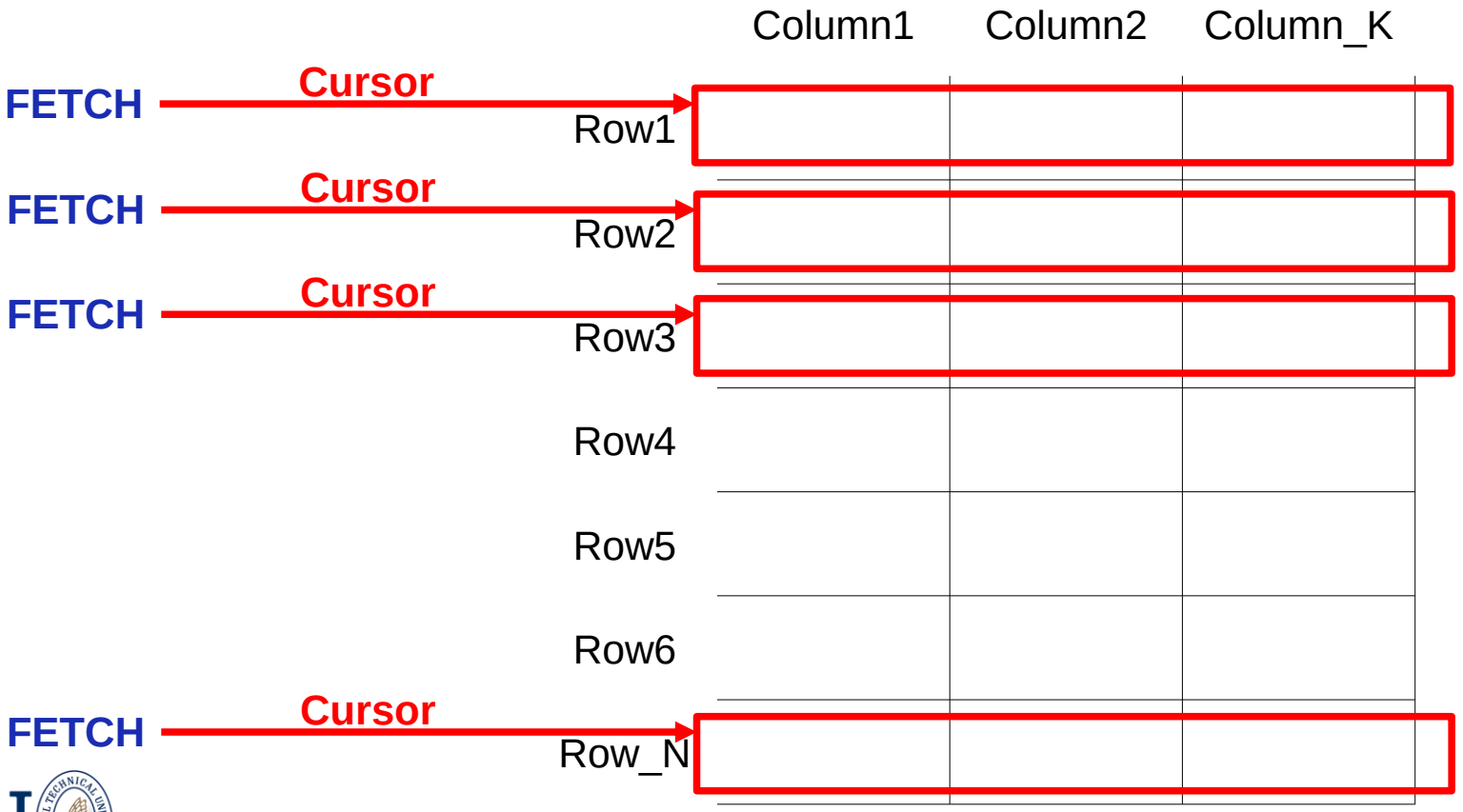
Step	Command	Description
1	`DECLARE cursor_name CURSOR FOR SELECT_statement;`	Cursor declaration is written first.
2	`DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished_flag = 1;`	• Declare a NOT FOUND handler to handle the situation when cursor could not find any row. • Each time you call the FETCH statement, the cursor attempts to read the next row in the result set. • When cursor reaches end of the result set, it will not be able to get the data, and a condition is raised. • The handler is used to handle this condition. • The finished_flag is a variable to indicate that the cursor has reached the end of result set.

Steps for Using a Cursor

Step	Command	Description
3	`OPEN cursor_name;`	OPEN statement initializes the result set for the cursor.
4	`FETCH cursor_name INTO variables_list;`	FETCH statement retrieves the next row pointed by the cursor and move the cursor to the next row in result set.
5	`IF finished_flag = 1 THEN ....`	Check if there is any row available before fetching it.
6	`CLOSE cursor_name;`	Close the cursor.

Fetching Rows of a Result-set with a Cursor

- A cursor is a reserved area in RAM memory, in which the output (result-set) of the SELECT query is stored, like an array holding columns and rows.
- Cursors are held in a reserved memory area in the DBMS server.
- Rows of result-set are retrieved by FETCH commands.



Example Table : CUSTOMER

```
CREATE TABLE customer (
    customer_id  INT,
    first_name   VARCHAR(20),
    last_name    VARCHAR(20),
    email        VARCHAR(50),
    CONSTRAINT PRIMARY KEY (customer_id)
);
```

```
INSERT INTO customer (customer_id, first_name, last_name,
                      email) VALUES
(1,  'Ursola',  'Purdy',     'upurdy0@cdnews.com'),
(2,  'Ruthe',   'Vatini',    'rvatini1@fema.gov'),
(3,  'Reidar',  'Turbitt',   'rturbitt2@geocities.jp'),
(4,  'Rich',    'Kirsz',     'rkirsz3@jalbum.net'),
(5,  'Carline', 'Kupis',     'ckupis4@tamu.edu'),
(6,  'Kandy',   'Adamec',    'kadamec5@weather.com'),
(7,  'Jermain', 'Giraudeau', 'jgiraudeau6@elpais.com'),
(8,  'Nolly',   'Bonicelli', 'nbonicelli7@examiner.com'),
(9,  'Phebe',   'Curdell',   'pcurdell8@usa.gov'),
(10, 'Euell',   'Guilder',   'eguilder9@themeforest.net'),
(11, 'Teriann', 'Marritt',   'tmarritta@va.gov'),
(12, 'Vilmer',  'Douse',     'vdouseb@foxnews.com') ;
```

Example: Stored Procedure using a Cursor

- The stored procedure below builds and returns a long list string via the INOUT parameter (**list** variable), which contains email addresses of all customers.
- The email addresses are separated by (;) characters.

Part1

```
DELIMITER $$
CREATE PROCEDURE generate_email_list (
                INOUT list varchar(2000) )
BEGIN
    DECLARE  status          INTEGER DEFAULT 0;
    DECLARE  email_address   varchar(100) DEFAULT "";

    -- declare cursor for result-set
    DECLARE table_cursor
        CURSOR FOR
            SELECT email FROM customer;

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

Example: Stored Procedure using a Cursor

Part2

```
    OPEN table_cursor;
    KeepOn: LOOP
        FETCH table_cursor INTO email_address;
        IF status = 1 THEN
            LEAVE KeepOn;
        END IF;
        -- build email list, separated by symbols
        SET list = CONCAT(list, " ; ",
                          email_address);
    END LOOP KeepOn;
    CLOSE table_cursor;
END$$
DELIMITER ;
```

Calling the stored procedure:

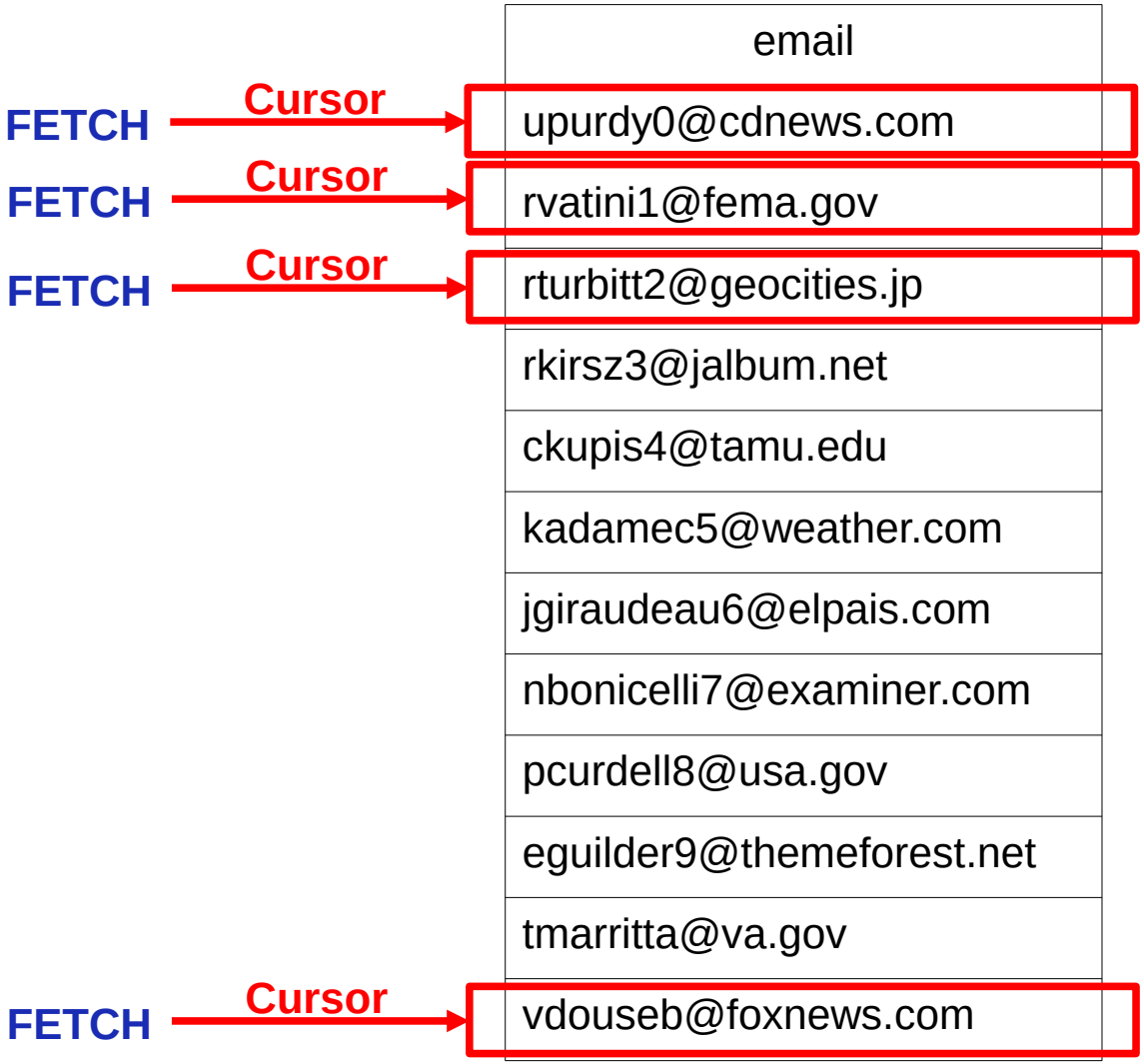
Queries

```
SET @list = "";

CALL generate_email_list (@list);

SELECT @list;
```

Result string
(list variable)

```
upurdy0@cdnews.com ; rvatini1@fema.gov ; rturbitt2@geocities.jp ;
rkirsz3@jalbum.net ; ckupis4@tamu.edu ; kadamec5@weather.com ;
jgiraudeau6@elpais.com ; nbonicelli7@examiner.com ; pcurdell8@usa.gov ;
eguilder9@themeforest.net ; tmarritta@va.gov ; vdouseb@foxnews.com ;
```

Fetching Rows of a Result-set with a Cursor



Topics

- Variables
- Built-in Functions
- Stored Functions
- Stored Procedures
- Procedural Language statements
- Random Data Generation
- Cursors
- Triggers

Triggers

- A trigger is a stored prstudentam on database, that runs automatically when an INSERT, UPDATE, or DELETE operation is executed.
- **Although a trigger is similar to a stored procedure, it can not be explicitly called.**
- It can only run when the event that the trigger is linked to is run.
- Trigger prstudentams are pre-compiled, and their executable codes are stored in database.
- A trigger is associated with a particular table and is defined to activate for INSERT, DELETE, or UPDATE statements for that table.
- The trigger definition includes a statement that executes, when the trigger activate

Trigger Usages

Example usage1:

- A trigger can examine or change new data values to be inserted or used to update a row.
- This enables you to enforce data integrity constraints.
- It also makes it possible to perform input data filtering.

Example usage2:

- A trigger can be used to perform logging (recording) of changes to existing rows in a table.

Trigger Syntax

Syntax for trigger creation :

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER}                          -- when the trigger activates
{INSERT | UPDATE | DELETE}                -- what statement activates it
ON table_name                             -- the associated table
FOR EACH ROW trigger_statements;          -- what the trigger does
```

Trigger Syntax

- **table_name** is the table with which the trigger is associated;
- **trigger_statements** is the trigger body, the statements that executes when the trigger activates.
- In a trigger body, the syntax NEW.column_name can be used to refer to columns in the new row to be inserted or updated in an INSERT or UPDATE trigger.
- Similarly, OLD.column_name can be used to refer to columns in the old row.
- To change a column value within a BEFORE trigger before the value is stored in the table, use following:

SET NEW.column_name = value.

Example Table

- First, a table is created with following command.
- Suppose valid range of values for the Data field must be 0 to 100.

```
create table TIME_TABLE
(
   Data        int,
   Date_Time   DATETIME
);
```

Example Trigger Prstudentam

- The following example shows a trigger prstudentam for INSERT statements for a table.
- The trigger uses BEFORE clause, so that it can examine data values of the Data field in table, before they are inserted into the table.
- The trigger performs two actions:
  - For attempts to insert a data value that lies outside the range from 0 to 100, the trigger converts the value to the nearest endpoint.
  - The trigger automatically provides a value of CURRENT_TIMESTAMP for the DATETIME column.

Example Trigger Prstudentam

```
delimiter $
CREATE TRIGGER  Trigger1
   BEFORE INSERT ON TIME_TABLE
   FOR EACH ROW
     BEGIN
       SET NEW.Date_Time = CURRENT_TIMESTAMP;

       IF NEW.Data < 0 THEN
          SET NEW.Data = 0;
       ELSEIF NEW.Data > 100 THEN
          SET NEW.Data = 100;
      END IF;
END$
delimiter ;
```

For each INSERT command below, the trigger prstudentam is called automatically.

```
INSERT INTO TIME_TABLE (Data) VALUES(-75); DO SLEEP(2);

INSERT INTO TIME_TABLE (Data) VALUES(36);  DO SLEEP(2);

INSERT INTO TIME_TABLE (Data) VALUES(0);   DO SLEEP(2);

INSERT INTO TIME_TABLE (Data) VALUES(120); DO SLEEP(2);

INSERT INTO TIME_TABLE (Data) VALUES(100); DO SLEEP(2);

INSERT INTO TIME_TABLE (Data) VALUES(44);  DO SLEEP(2);
```

Query

```
SELECT *
FROM
TIME_TABLE;
```

Result-set

Data	Date_Time
0	2023-10-25 12:03:28
36	2023-10-25 12:03:30
0	2023-10-25 12:03:32
100	2023-10-25 12:03:34
100	2023-10-25 12:03:36
44	2023-10-25 12:03:38