

Module 7

# Advanced SQL – Stored Procedures

# Stored Procedures without Variables

- A stored procedure is a pre-prepared SQL code that you can save, so the code can be reused over and over again.
- Stored procedure can hold one or a group of SQL statements;

```
DELIMITER $$
```

```
CREATE PROCEDURE Procedure_Name()
```


```
BEGIN
```

```
    SQL_QUERY;
```


```
END $$
```

```
DELIMITER ;
```

```
CALL Procedure_Name();
```



# Stored Procedures without Variables

- Create Procedure is the command..
  - The green colors (Create Procedure, (), BEGIN and END) are the syntax for the stored procedures and do not change.
  - The yellow colors (Procedure\_Name & SQL\_Statement) changes per use case.
  - CALL executes the stored procedure.
- 

# Delimiter

- Delimiter changes the default End of Command from ; to the specified \$\$\$. We may use any character.
- A MySQL client program such as MySQL Workbench or mysql program uses the delimiter (;) to separate statements and execute each separately.
- Since we need to use “;” in the stored SQL statement, we need to temporarily change the delimiter.
- After the stored procedure we change it back to “;” using the DELIMITER command.

# Stored Procedures without Variables

**DELIMITER \$\$** --← We have to change the delimiter temporarily so that the default delimiter ; can be use in the stored SQL code

**CREATE PROCEDURE** **select\_all()** --← Name of the stored Procedure. Yellow may be modified as per use case.

**BEGIN** --← Begin the Procedure

**select faculty\_name from faculty\_table**

**where faculty\_id = 1001;** --← (Remember ; is not the delimiter. Currently the above line is simply stored as a code)

**END \$\$** --← End stored procedure, use the new delimiter \$\$ to signal end of procedure.

**DELIMITER ;** --← Set the default delimiter back to ;

# Stored Procedures

- Step one: We change the delimiter to \$\$ (you may use any character)
- Step two: We use the command `create_procedure` followed by the procedure name `()`

(Note: you may use any name you want as the procedure name. Don't forget the `()` at the end of the procedure name)

- Step three : Begin procedure and type the required SQL Query including the default delimiter `;` (Since we changed the delimiter, it wont register as an end of command here)
- Step four: End the procedure
- Step five: Change delimiter back to `;` so other statements wont throw errors.
- Step six: Call the procedure.

```
1  USE CH06_ICQ;
2
3  DELIMITER $$
4  * create procedure select_all()
5  ⊖ BEGIN
6      select faculty_name from faculty_table where faculty_id = 1001;
7  END $$
8  DELIMITER ;
9
10 * Call select_all();
11
12 * drop procedure if exists select_all;
13
14
```

100% 19:10

Result Grid



Filter Rows:



Search

Export:



faculty_name
Johnson

# Stored Procedures with Variables

```
DELIMITER $$
```

```
CREATE PROCEDURE Procedure_Name()
```

```
BEGIN
```

```
    DECLARE Variable_Name, Variable Type,  
    DEFAULT Value;
```

```
    SQL_QUERY;
```

```
END $$
```

```
DELIMITER ;
```





# Stored Procedures with Variables

**DELIMITER \$\$** --← We have to change the delimiter temporarily so that the default delimiter ; can be use in the stored SQL code.

**CREATE PROCEDURE** GetTotalFaculty() --← Name of the stored Procedure. Yellow may be modified as per use case.

**BEGIN** ← Begin the Procedure

**DECLARE** totalfaculty **INT** **DEFAULT** 0; --← Declaring variable totalfaculty as type INT with default value 0

**SELECT** COUNT(\*) --← you may select anything you wish to display, or use any code

**INTO** totalfaculty

**FROM** faculty\_table; --← We use ';' here as part of the SQL code. This is why we had to change the delimiter earlier. If not it would consider this ';' as the end of code.

**SELECT** totalfaculty; --← Prints totalfaculty later when the stored procedure is called.

**END\$\$** --← Remember \$\$ is the new delimiter, so this is considered the official end of SQL stored procedure code (SS replaced ;)

**DELIMITER** ; --← We have to change the delimiter back to ;

**Call** GetTotalFaculty(); --← Here we use the default delimiter after calling the stored procedure.

```
14 DELIMITER $$
15 CREATE PROCEDURE GetTotalFaculty()
16 BEGIN
17     DECLARE totalfaculty INT DEFAULT 0;
18
19     SELECT COUNT(*)
20     INTO totalfaculty
21     FROM faculty_table;
22
23     SELECT totalfaculty;
24 END$
25 DELIMITER ;
26
27 Call GetTotalFaculty();
28
```

100%



1:12

Result Grid



Filter Rows:



Search

Export:



tot...
3

# SQL Code Breakdown

- We use the declare command to declare variables.
- `DECLARE totalfaculty INT DEFAULT 0;`
  - Here we declare a variable called totalfaculty
  - Its datatype is INT
  - Default value is 0
- `SELECT COUNT(*) INTO totalfaculty FROM faculty_table;`
  - This SQL command selects the total count from the faculty table and inserts into the variable totalfaculty.
- `SELECT totalfaculty;`
  - Selects totalfaculty to print when we call the stored procedure later.

# IF – ELSE Statements

- The IF-THEN statement allows you to execute a set of SQL statements based on a specified condition.
- IF condition THEN  
statements;  
ELSE ← Else statements are optional. If can work without else.  
else-statements;  
END IF;
- The IF statement can be included with our regular stored procedures

# IF Statement

- The IF statement executes statements if the specified condition is true.
- Here we want to check if the totalfaculty is greater than 1.
- Totalfaculty counts the instances in the faculty\_table.
- In our case totalfaculty is 3. Which is greater than 1. The condition is true, and we print totalfaculty.
- You can see the output as 3 below.

```
3 DELIMITER $$
4 CREATE PROCEDURE GetTotalFaculty()
5 BEGIN
6     DECLARE totalfaculty INT DEFAULT 0;
7
8     SELECT COUNT(*)
9     INTO totalfaculty
10    FROM FACULTY_TABLE;
11
12    IF totalfaculty > 1 THEN
13        SELECT totalfaculty;
14    END IF;
15 END $$
16
17 CALL GetTotalFaculty();
```

100% 24:17

Result Grid Filter Rows: Search Export:

tot...
3

# IF - ELSE

- Here we modify the previous code.
- We check if totalfaculty is less than 1. We already know totalfaculty is 3. So let us test the else statement.
- The else statement should activate if the condition is false.
- In our case the condition was false, so we select \* (everything) from the faculty table.
- You can see the output below.

```
14 DELIMITER $$
15 CREATE PROCEDURE GetAllFaculty()
16 BEGIN
17     DECLARE totalfaculty INT DEFAULT 0;
18
19     SELECT COUNT(*)
20     INTO totalfaculty
21     FROM faculty_table;
22
23     IF totalfaculty < 1 THEN
24         SELECT totalfaculty;
25     ELSE
26         SELECT * FROM Faculty_Table;
27     END IF;
28 END $$
29 DELIMITER ;
30
31 Call GetAllFaculty();
```

00% 22:31

Result Grid Filter Rows: Search Export:

faculty_id	faculty_name
1001	Johnson
1010	George
1020	William

Result 3

# LOOP STATEMENT

- LOOP statement runs a block of code repeatedly based on a condition.

```
[begin_label:] LOOP  
    statement_list;  
END LOOP
```

- The LOOP executes the statement\_list repeatedly.
- The statement\_list may have one or more statements, each terminated by a semicolon (;) statement delimiter.
- Typically, you terminate the loop when a condition is satisfied by using the LEAVE statement.
- We use Label to begin and end the loop block.

# LOOP STATEMENT with LEAVE

```
[label]: LOOP  
IF condition THEN  
    LEAVE [label];  
END IF;  
END LOOP;
```

- Here the loop exits, if the condition is true
- Leave exits the loop similar to break statement in other programming languages like PHP, C/C++, and Java.



# Loop Demo

```
DELIMITER $$ -- Change Delimiter to $$, we are now free to use ; in our stored code
CREATE PROCEDURE LoopDemo() -- Creating a new stored procedure called LoopDemo
BEGIN -- Begin the stored procedure
    DECLARE x INT; -- Declaring variable x as INT
    DECLARE str VARCHAR(255); -- Declaring another str variable as VARCHAR
    SET x = 1; -- Set value of x to 1.
    SET str = ' '; -- Set str as ' ' (blank space).
    loop_label: LOOP ← We now begin the loop block
        IF x > 10 THEN
            LEAVE loop_label; ← Exit the loop if x > 10
        END IF; ← ends the IF statement if x > 10
        SET x = x + 1; ← When x not greater than 1, increment x by 1
        IF (x mod 2) THEN ← Check if x is divisible by 2.
            ITERATE loop_label; ← If x is divisible by 2, repeat the loop
        ELSE
            SET str = CONCAT(str,x,','); ← If X is not divisible by 2
                                         concatenate str and x
        END IF; ← Ends the (x mod 2) if statement
    END LOOP; ← Ends the loop
    SELECT str; ← Prints str as the final output
END $$ ← Ends the procedure
DELIMITER ; ← Change default delimiter back
```

# Loop Demo & Code Breakdown

- We change delimiter to \$\$\$. Now we are free to use ; as the delimiter in our stored code.
- We create a stored procedure called LoopDemo
- We declare some variables required for the code
- We exit the loop if  $x > 10$ . If not, we may be in an infinite loop
- We keep increasing the value of  $x$  by 1 and keep checking if the new value of  $x$  is divisible by 2
- When true we keep concat (concatenating or adding on, the value of  $x$  to the string  $str$ ).
- Finally,  $str$  should contain all the values that are divisible by 2 in the range (1 to 10)
- We end the IF and the loop. We also set the delimiter back to ;
- We call the procedure and the output shows  $str$  (2,4,6,8,10)

```
35 DELIMITER $$$
36 CREATE PROCEDURE LoopDemo()
37 BEGIN
38     DECLARE x INT;
39     DECLARE str VARCHAR(255);
40
41     SET x = 1;
42     SET str = '';
43
44     loop_label: LOOP
45         IF x > 10 THEN
46             LEAVE loop_label;
47         END IF;
48
49         SET x = x + 1;
50         IF (x mod 2) THEN
51             ITERATE loop_label;
52         ELSE
53             SET str = CONCAT(str,x,',');
54         END IF;
55     END LOOP;
56     SELECT str;
57 END$$$
58
59 DELIMITER ;
60 CALL LoopDemo();
61
```

100% 1:60

Result Grid Filter Rows: Search Export:

str
2,4,6,8,10,

# Additional Resources

- <https://www.softwaretestinghelp.com/mysql-stored-procedure/>
- <https://www.youtube.com/watch?v=NrBJmtD0kEw>
- <https://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx>

The background features a gradient from light blue on the left to light green on the right. In the top-left corner, there are several overlapping, wavy, semi-transparent shapes in shades of blue and white. In the bottom-right corner, there are similar wavy shapes in shades of green and white.

END