Stored procedures: -

Define:

A stored procedure is a segment of declarative SQL statements stored inside the database catalog. A stored procedure can be invoked by [triggers](http://www.mysqltutorial.org/mysql-triggers.aspx), other stored procedures, and applications such as [Java](http://www.mysqltutorial.org/calling-mysql-stored-procedures-from-jdbc/), [Python](http://www.mysqltutorial.org/calling-mysql-stored-procedures-python/), [PHP](http://www.mysqltutorial.org/php-calling-mysql-stored-procedures/), etc.\

-> MySQL stored procedures are compiled on demand. After compiling a stored procedure, MySQL puts it into a cache. And MySQL maintains its own stored procedure cache for every single connection. If an application uses a stored procedure multiple times in a single connection, the compiled version is used, otherwise, the stored procedure works like a query.

* Stored procedures help reduce the traffic between application and database server because instead of sending multiple lengthy SQL statements, the application has to send only name and parameters of the stored procedure.
* Stored procedures are reusable and transparent to any applications. Stored procedures expose the database interface to all applications so that developers don’t have to develop functions that are already supported in stored procedures.
* Stored procedures are secure. The database administrator can grant appropriate permissions to applications that access stored procedures in the database without giving any permissions on the underlying database tables.

Let’s examine the stored procedure in greater detail:

* The first command is DELIMITER // , which is not related to the stored procedure syntax. The DELIMITER statement changes the standard delimiter which is semicolon ( ; ) to another. In this case, the delimiter is changed from the semicolon( ; ) to double-slashes // Why do we have to change the delimiter? Because we want to pass the stored procedure to the server as a whole rather than letting mysql tool interpret each statement at a time.  Following the END keyword, we use the delimiter //  to indicate the end of the stored procedure. The last command ( DELIMITER; *)* changes the delimiter back to the semicolon (;).
* We use the CREATE PROCEDURE  statement to create a new stored procedure. We specify the name of stored procedure after the CREATE PROCEDURE  statement. In this case, the name of the stored procedure is GetAllProducts *.*We put the parentheses after the name of the stored procedure*.*
* The section between BEGIN and END  is called the body of the stored procedure. You put the declarative SQL statements in the body to handle business logic. In this stored procedure, we use a simple [SELECT](http://www.mysqltutorial.org/mysql-select-statement-query-data.aspx)statement to query data from the products table.
* In workbench, how we can create a stored procedures are follow,
* Step1: right click on routines and select create procedure option
* Step2: write stored procedure code,

|  |
| --- |
| * DELIMITER // * CREATE PROCEDURE GetAllProducts() * BEGIN * SELECT \*  FROM products; * END // * DELIMITER ; * now new stored procedures is created |

**Calling stored procedures:**

You use the CALL  statement to call a stored procedure e.g., to call the GetAllProducts() stored procedure, you use the following statement:

CALL GetAllProducts();

**Stored procedure variables:**

 variable is a named data object whose value can change during the stored procedure execution. We typically use the variables in [stored procedures](http://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx) to hold the immediate results. These variables are local to the stored procedure.

->You must declare a variable before you can use it.

|  |  |
| --- | --- |
| 1 | DECLARE variable\_name datatype(size) DEFAULT default\_value; |

For example, we can declare a variable named   total\_sale with the data type INT and default value 0  as follows:



|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1 | DECLARE total\_sale INT DEFAULT 0; | |

MySQL allows you to declare two or more variables that share the same data type using a single DECLARE statement as following:



|  |  |
| --- | --- |
| 1 | **DECLARE x, y INT DEFAULT 0;** |

We declared two integer variables  x and  y, and set their default values to zero.

## Assigning variables

Once you declared a variable, you can start using it. To assign a variable another value, you use the SET  statement, for example:



|  |  |
| --- | --- |
| 1  2 | **DECLARE total\_count INT DEFAULT 0;**  **SET total\_count = 10;** |

The value of the total\_count variable is 10  after the assignment.

Besides the SET  statement, you can use the SELECT INTO  statement to assign the result of a query, which returns a scalar value, to a variable.

|  |  |
| --- | --- |
| 1  2  3  4 | DECLARE total\_products INT DEFAULT 0    SELECT COUNT(\*) INTO total\_products  FROM products |

**Mysql stored procedure parameters:**

* IN – is the default mode. When you define an IN parameter in a stored procedure, the calling program has to pass an argument to the stored procedure. In addition, the value of an IN parameter is protected. It means that even the value of the IN parameter is changed inside the stored procedure, its original value is retained after the stored procedure ends. In other words, the stored procedure only works on the copy of the IN parameter.
* Example,

### The IN parameter example

The following example illustrates how to use the IN parameter in the GetOfficeByCountrystored procedure that selects offices located in a particular country.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | DELIMITER //  CREATE PROCEDURE GetOfficeByCountry(IN countryName VARCHAR(255))  BEGIN  SELECT \*  FROM offices  WHERE country = countryName;  END //  DELIMITER ; |

The countryName is the IN parameter of the stored procedure. Inside the stored procedure, we select all offices that locate in the country specified by the countryName parameter.

Suppose, we want to get all offices in the USA, we just need to pass a value (USA) to the stored procedure as follows:



|  |  |
| --- | --- |
| 1 | **CALL GetOfficeByCountry('USA');** |

* OUT – the value of an OUT parameter can be changed inside the stored procedure and its new value is passed back to the calling program. Notice that the stored procedure cannot access the initial value of the OUT parameter when it starts.

### The OUT parameter example

The following stored procedure returns the number of orders by order status. It has two parameters:

* orderStatus : the IN parameter that is the order status which we want to count the orders.
* total : the OUT parameter that stores the number of orders for a specific order status.

The following is the source code of the CountOrderByStatus stored procedure.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | DELIMITER $$  CREATE PROCEDURE CountOrderByStatus(  IN orderStatus VARCHAR(25),  OUT total INT)  BEGIN  SELECT count(orderNumber)  INTO total  FROM orders  WHERE status = orderStatus;  END$$  DELIMITER ; |

To get the number of shipped orders, we call the CountOrderByStatus  stored procedure and pass the order status as Shipped, and also pass an argument ( @total ) to get the return value.



|  |  |
| --- | --- |
| 1  2 | CALL CountOrderByStatus('Shipped',@total);  SELECT @total; |

http://www.mysqltutorial.org/wp-content/uploads/2009/12/MySQL-OUT-parameter-order-shipped.png

To get the number of orders that are in process, we call the CountOrderByStatus stored procedure as follows:



|  |  |
| --- | --- |
| 1  2 | CALL CountOrderByStatus('in process',@total);  SELECT @total AS  total\_in\_process; |

http://www.mysqltutorial.org/wp-content/uploads/2009/12/MySQL-OUT-parameter-orders-in-process.png

* INOUT – an INOUT  parameter is the combination of IN  and OUT  parameters. It means that the calling program may pass the argument, and the stored procedure can modify the INOUT parameter and pass the new value back to the calling program.

### The INOUT parameter example

The following example demonstrates how to use an INOUT parameter in the stored procedure.



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | DELIMITER $$  CREATE PROCEDURE set\_counter(INOUT count INT(4),IN inc INT(4))  BEGIN  SET count = count + inc;  END$$  DELIMITER ; |

How it works.

* The set\_counter  stored procedure accepts one INOUT  parameter ( count ) and one IN parameter ( inc ).
* Inside the stored procedure, we increase the counter ( count ) by the value of the incparameter.

See how we call the set\_counter  stored procedure:



|  |  |
| --- | --- |
| 1  2  3  4  5 | SET @counter = 1;  CALL set\_counter(@counter,1); -- 2  CALL set\_counter(@counter,1); -- 3  CALL set\_counter(@counter,5); -- 8  SELECT @counter; - |

The syntax of defining a parameter in the stored procedures is as follows:



|  |  |
| --- | --- |
| 1 | **MODE param\_name param\_type(param\_size)** |

**Stored procedures that returns multiple values:**

[MySQL stored function](http://www.mysqltutorial.org/mysql-stored-function/) returns only one value. To develop stored programs that return multiple values, you need to use [stored procedures](http://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx) with INOUT or OUT parameters.

DELIMITER $$  
   
CREATE PROCEDURE get\_order\_by\_cust(  
IN cust\_no INT,  
OUT shipped INT,  
OUT canceled INT,  
OUT resolved INT,  
OUT disputed INT)  
BEGIN  
-- shipped  
SELECT  
            count(\*) INTO shipped  
        FROM  
            orders  
        WHERE  
            customerNumber = cust\_no  
                AND status = 'Shipped';  
   
-- canceled  
SELECT  
            count(\*) INTO canceled  
        FROM  
            orders  
        WHERE  
            customerNumber = cust\_no  
                AND status = 'Canceled';  
   
-- resolved  
SELECT  
            count(\*) INTO resolved  
        FROM  
            orders  
        WHERE  
            customerNumber = cust\_no  
                AND status = 'Resolved';  
   
-- disputed  
SELECT  
            count(\*) INTO disputed  
        FROM  
            orders  
        WHERE  
            customerNumber = cust\_no  
                AND status = 'Disputed';  
   
END

IF ELSEIF STATEMENTS:

DELIMITER $$  
   
CREATE PROCEDURE GetCustomerLevel(  
    in  p\_customerNumber int(11),   
    out p\_customerLevel  varchar(10))  
BEGIN  
    DECLARE creditlim double;  
   
    SELECT creditlimit INTO creditlim  
    FROM customers  
    WHERE customerNumber = p\_customerNumber;  
   
    IF creditlim > 50000 THEN  
SET p\_customerLevel = 'PLATINUM';  
    ELSEIF (creditlim <= 50000 AND creditlim >= 10000) THEN  
        SET p\_customerLevel = 'GOLD';  
    ELSEIF creditlim < 10000 THEN  
        SET p\_customerLevel = 'SILVER';  
    END IF;  
   
END$$

**CASE STATEMENTS:**

Besides the [IF statement](http://www.mysqltutorial.org/mysql-if-statement/), MySQL provides an alternative conditional statement called CASE. The MySQL CASE statement makes the code more readable and efficient.

There are two forms of the CASE statements: simple and searched CASE statements

DELIMITER $$  
   
CREATE PROCEDURE GetCustomerShipping(  
in  p\_customerNumber int(11),   
out p\_shiping        varchar(50))  
BEGIN  
    DECLARE customerCountry varchar(50);  
   
    SELECT country INTO customerCountry  
FROM customers  
WHERE customerNumber = p\_customerNumber;  
   
    CASE customerCountry  
WHEN  'USA' THEN  
   SET p\_shiping = '2-day Shipping';  
WHEN 'Canada' THEN  
   SET p\_shiping = '3-day Shipping';  
ELSE  
   SET p\_shiping = '5-day Shipping';  
END CASE;  
   
END$$

WHILE STATEMENT:

DELIMITER $$  
DROP PROCEDURE IF EXISTS test\_mysql\_while\_loop$$  
CREATE PROCEDURE test\_mysql\_while\_loop()  
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 ;

REPEAT STATEMENT:

DELIMITER $$  
DROP PROCEDURE IF EXISTS mysql\_test\_repeat\_loop$$  
CREATE PROCEDURE mysql\_test\_repeat\_loop()  
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 ;

LOOP, LEAVE and ITERATE statements

There are two statements that allow you to control the loop:

* The LEAVE statement allows you to exit the loop immediately without waiting for checking the condition. The LEAVE statement works like the  break statement in other languages such as PHP, C/C++, Java, etc.
* The ITERATE statement allows you to skip the entire code under it and start a new iteration. The ITERATE statement is similar to the continuestatement in PHP, C/C++, Java, etc.
* CREATE PROCEDURE test\_mysql\_loop()  
  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;

**MYSQL CURSOR:**

DELIMITER $$  
   
CREATE PROCEDURE build\_email\_list (INOUT email\_list varchar(4000))  
BEGIN  
   
DECLARE v\_finished INTEGER DEFAULT 0;  
        DECLARE v\_email varchar(100) DEFAULT "";  
   
-- declare cursor for employee email  
DEClARE email\_cursor CURSOR FOR   
SELECT email FROM employees;  
   
-- declare NOT FOUND handler  
DECLARE CONTINUE HANDLER   
        FOR NOT FOUND SET v\_finished = 1;  
   
OPEN email\_cursor;  
   
get\_email: LOOP  
   
FETCH email\_cursor INTO v\_email;  
   
IF v\_finished = 1 THEN   
LEAVE get\_email;  
END IF;  
   
-- build email list  
SET email\_list = CONCAT(v\_email,";",email\_list);  
   
END LOOP get\_email;  
   
CLOSE email\_cursor;  
   
END$$  
   
DELIMITER ;

**SIGNAL STATEMENT:**

DELIMITER $$  
   
CREATE PROCEDURE AddOrderItem(  
         in orderNo int,  
in productCode varchar(45),  
in qty int,   
                         in price double,   
                         in lineNo int )  
BEGIN  
DECLARE C INT;  
   
SELECT COUNT(orderNumber) INTO C  
FROM orders   
WHERE orderNumber = orderNo;  
   
-- check if orderNumber exists  
IF(C != 1) THEN   
SIGNAL SQLSTATE '45000'  
SET MESSAGE\_TEXT = 'Order No not found in orders table';  
END IF;  
-- more code below  
-- ...  
END

**Join sql:**

## SQL INNER JOIN Keyword

The INNER JOIN keyword selects all rows from both tables as long as there is a match between the columns in both tables.

### SQL INNER JOIN Syntax

SELECT column\_name(s)  
FROM table1  
INNER JOIN table2  
ON table1.column\_name=table2.column\_name;

or:

SELECT column\_name(s)  
FROM table1  
JOIN table2  
ON table1.column\_name=table2.column\_name;

## SQL LEFT JOIN Keyword

The LEFT JOIN keyword returns all rows from the left table (table1), with the matching rows in the right table (table2). The result is NULL in the right side when there is no match.

### SQL LEFT JOIN Syntax

SELECT column\_name(s)  
FROM table1  
LEFT JOIN table2  
ON table1.column\_name=table2.column\_name;

or:

SELECT column\_name(s)  
FROM table1  
LEFT OUTER JOIN table2  
ON table1.column\_name=table2.column\_name;

## SQL RIGHT JOIN Keyword

The RIGHT JOIN keyword returns all rows from the right table (table2), with the matching rows in the left table (table1). The result is NULL in the left side when there is no match.

### SQL RIGHT JOIN Syntax

SELECT column\_name(s)  
FROM table1  
RIGHT JOIN table2  
ON table1.column\_name=table2.column\_name;

or:

SELECT column\_name(s)  
FROM table1  
RIGHT OUTER JOIN table2  
ON table1.column\_name=table2.column\_name;

FULL OUTER JOINS:

The SQL **FULL JOIN** combines the results of both left and right outer joins.

The joined table will contain all records from both tables, and fill in NULLs for missing matches on either side.

## Syntax:

The basic syntax of **FULL JOIN** is as follows:

SELECT table1.column1, table2.column2... FROM table1 FULL JOIN table2 ON table1.common\_field = table2.common\_field;

**TRIGGERS:**

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events:

* A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
* A database definition (DDL) statement (CREATE, ALTER, or DROP).
* A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers could be defined on the table, view, schema, or database with which the event is associated.

CREATE OR REPLACE TRIGGER display\_salary\_changes BEFORE DELETE OR INSERT OR UPDATE ON customers FOR EACH ROW WHEN (NEW.ID > 0) DECLARE sal\_diff number; BEGIN sal\_diff := :NEW.salary - :OLD.salary; dbms\_output.put\_line('Old salary: ' || :OLD.salary); dbms\_output.put\_line('New salary: ' || :NEW.salary); dbms\_output.put\_line('Salary difference: ' || sal\_diff); END;