# MySQL Subquery

A subquery in MySQL is a query, which is nested into another SQL query and embedded with SELECT, INSERT, UPDATE or DELETE statement along with the various operators. We can also nest the subquery with another subquery. A subquery is known as the **inner query**, and the query that contains subquery is known as the **outer query**. The inner query executed first gives the result to the outer query, and then the main/outer query will be performed.

[MySQL](https://www.javatpoint.com/mysql-tutorial) allows us to use subquery anywhere, but it must be closed within parenthesis. All subquery forms and operations supported by the SQL standard will be supported in MySQL also.

**The following are the rules to use subqueries:**

* Subqueries should always use in **parentheses.**
* If the main query does not have multiple columns for subquery, then a subquery can have only one column in the SELECT command.
* We can use various comparison operators with the subquery, such as >, <, =, IN, ANY, SOME, and ALL. A multiple-row operator is very useful when the subquery returns more than one row.
* We cannot use the **ORDER BY** clause in a subquery, although it can be used inside the main query.
* If we use a subquery in a **set function**, it cannot be immediately enclosed in a set function.

**The following are the advantages of using subqueries:**

* The subqueries make the queries in a structured form that allows us to isolate each part of a statement.
* The subqueries provide alternative ways to query the data from the table; otherwise, we need to use complex joins and unions.
* The subqueries are more readable than complex join or union statements.

## **MySQL Subquery Syntax**

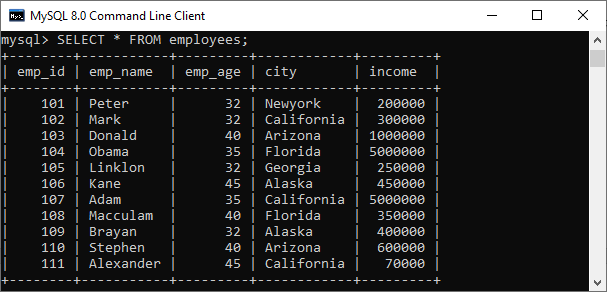
The following is the basic syntax to use the subquery in MySQL:

1. **SELECT** column\_list (s) **FROM**  table\_name
2. **WHERE**  column\_name OPERATOR
3. (**SELECT** column\_list (s)  **FROM** table\_name [**WHERE**])

## **MySQL Subquery Example**

Let us understand it with the help of an example. Suppose we have a table named **"employees"** that contains the following data:

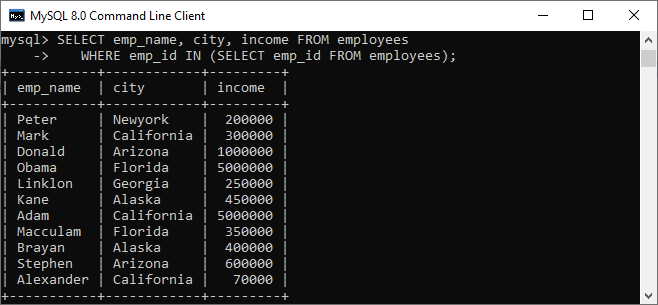
**Table: employees**



Following is a simple SQL statement that returns the **employee detail whose id matches in a subquery**:

1. **SELECT** emp\_name, city, income **FROM** employees
2. **WHERE** emp\_id IN (**SELECT** emp\_id **FROM** employees);

This query will return the following output:



### MySQL Subquery with Comparison Operator

A comparison operator is an operator used to compare values and returns the result, either true or false. The following comparison operators are used in MySQL <, >, =, <>, <=>, etc. We can use the subquery before or after the comparison operators that return a single value. The returned value can be the arithmetic expression or a column function. After that, SQL compares the subquery results with the value on the other side of the comparison operator. The below example explains it more clearly:

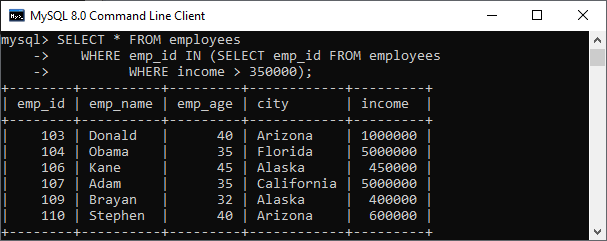
Following is a simple [SQL](https://www.javatpoint.com/sql-tutorial)

statement that returns the **employee detail whose income is more than 350000** with the help of subquery:

1. **SELECT** \* **FROM** employees
2. **WHERE** emp\_id IN (**SELECT** emp\_id **FROM** employees
3. **WHERE** income > 350000);

This query first executes the subquery that returns the **employee id whose income > 350000**. Second, the main query will return the employees all details whose employee id are in the result set returned by the subquery.

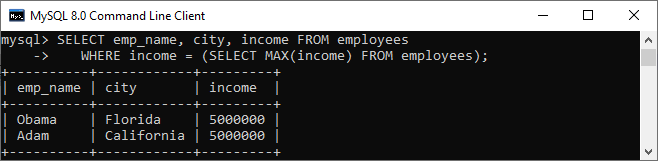
After executing the statement, we will get the below output, where we can see the employee detail whose income>350000.



Let us see an example of another comparison operator, such as equality (=) to find employee details with **maximum income** using a subquery.

1. **SELECT** emp\_name, city, income **FROM** employees
2. **WHERE** income = (**SELECT** **MAX**(income) **FROM** employees);

It will give the output where we can see two employees detail who have maximum income.

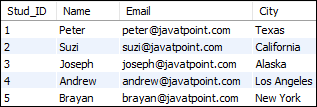


### MySQL Subquery with IN or NOT-IN Operator

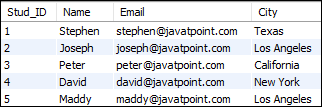
If the subquery produces more than one value, we need to use the IN or NOT IN operator with the [WHERE clause](https://www.javatpoint.com/mysql-where)

. Suppose we have a table named **"Student"** and **"Student2"** that contains the following data:

**Table: Student**



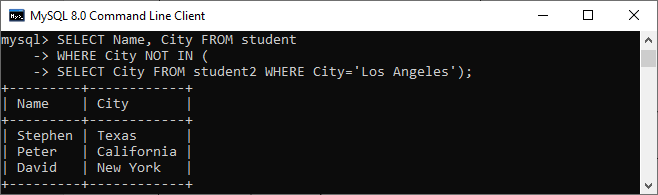
**Table: Student2**



The following subquery with NOT IN operator returns the **student detail who does not belong to Los Angeles City** from both tables as follows:

1. **SELECT** **Name**, City **FROM** student
2. **WHERE** City NOT IN (
3. **SELECT** City **FROM** student2 **WHERE** City='Los Angeles');

After execution, we can see that the result contains the student details not belonging to Los Angeles City.



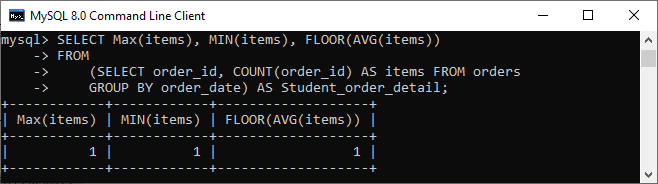
### MySQL Subquery in the FROM Clause

If we use a subquery in the FROM clause, MySQL will return the output from a subquery is used as a temporary table. We called this table as a derived table, inline views, or materialized subquery.

The following subquery returns the maximum, minimum, and average number of items in the order table:

1. **SELECT** **Max**(items), **MIN**(items), FLOOR(AVG(items))
2. **FROM**
3. (**SELECT** order\_id, COUNT(order\_id) **AS** items **FROM** orders
4. **GROUP** **BY** order\_date) **AS** Student\_order\_detail;

It will give the output as follows:



Ex: SELECT MAX(C),MIN(C),FLOOR(AVG(C)) FROM (SELECT DEPTNO,COUNT(DEPTNO) AS C FROM EMP GROUP BY JOB) AS RESULT;

### MySQL Correlated Subqueries

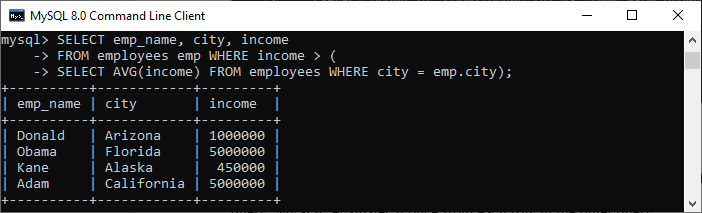
A correlated subquery in MySQL is a subquery that depends on the outer query. It uses the data from the outer query or contains a reference to a parent query that also appears in the outer query. MySQL evaluates it once from each row in the outer query.

**SELECT** emp\_name, city, income

**FROM** employees emp **WHERE** income > (

**SELECT** AVG(income) **FROM** employees **WHERE** city = emp.city);

In the above query, we select an **employee name and city** whose income is higher than the average income of all employees in each city.



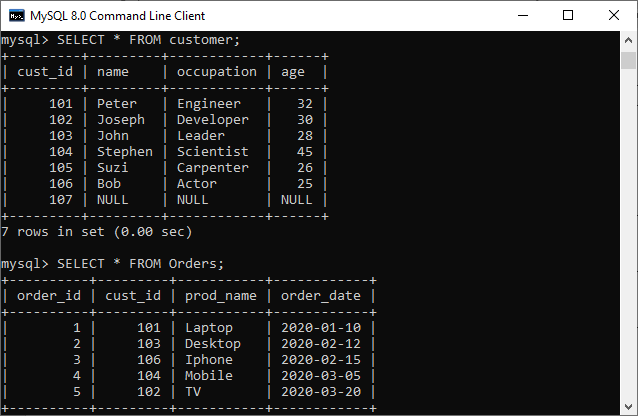
The subquery executes for every city of the specified table because it will change for every row. Therefore, the average income will also be changed. Then, the main query filters employee detail whose income is higher than the average income from the subquery.

Ex: SELECT ENAME,SAL FROM EMP E WHERE SAL>(SELECT AVG(SAL) FROM EMP WHERE DEPTNO=E.DEPTNO);

### MySQL Subqueries with EXISTS or NOT EXISTS

The [EXISTS operator](https://www.javatpoint.com/mysql-exists)

is a Boolean operator that returns either true or false result. It is used with a subquery and checks the existence of data in a subquery. If a subquery returns any record at all, this operator returns true. Otherwise, it will return false. The NOT EXISTS operator used for negation that gives true value when the subquery does not return any row. Otherwise, it returns false. Both EXISTS and NOT EXISTS used with correlated subqueries. The following example illustrates it more clearly. Suppose we have a table **customer and order** that contains the data as follows:



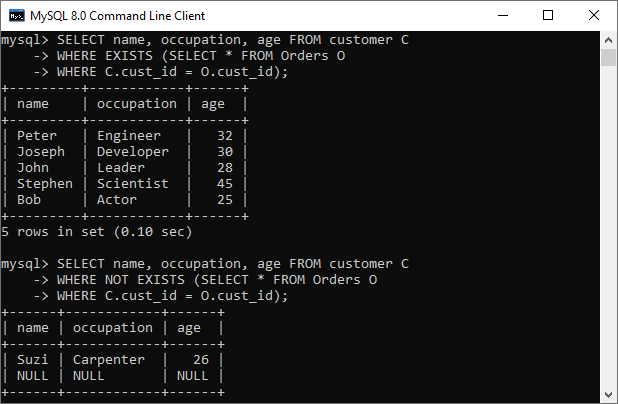
The below SQL statements uses EXISTS operator to find the name, occupation, and age of the customer who has placed at least one order.

1. **SELECT** **name**, occupation, age **FROM** customer C
2. **WHERE** EXISTS (**SELECT** \* **FROM** Orders O
3. **WHERE** C.cust\_id = O.cust\_id);

This statement uses NOT EXISTS operator that returns the customer details who have not placed an order.

1. **SELECT** **name**, occupation, age **FROM** customer C
2. **WHERE** NOT EXISTS (**SELECT** \* **FROM** Orders O
3. **WHERE** C.cust\_id = O.cust\_id);

We can see the below output to understand the above queries result.



.Ex:

SELECT DNAME,LOC FROM DEPT D WHERE EXISTS(SELECT \*FROM EMP E WHERE E.DEPTNO=D.DEPTNO);

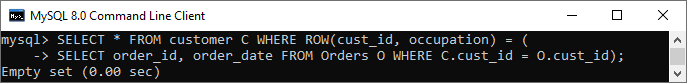
SELECT DNAME,LOC FROM DEPT D WHERE NOT EXISTS(SELECT \*FROM EMP E WHERE E.DEPTNO=D.DEPTNO);

### MySQL ROW Subqueries

It is a subquery that returns a single row where we can get more than one column values. We can use the following operators for comparing row subqueries =, >, <, >=, <=, <>, !=, <=>. Let us see the following example:

1. **SELECT** \* **FROM** customer C **WHERE** ROW(cust\_id, occupation) = (
2. **SELECT** order\_id, order\_date **FROM** Orders O **WHERE** C.cust\_id = O.cust\_id);

If given row has cust\_id, occupation values equal to the order\_id, order\_date values of any rows in the first table, the WHERE expression is TRUE, and each query returns those first table rows. Otherwise, the expression is FALSE, and the query produces an empty set, which can be shown in the below image:



### MySQL Subqueries with ALL, ANY, and SOME

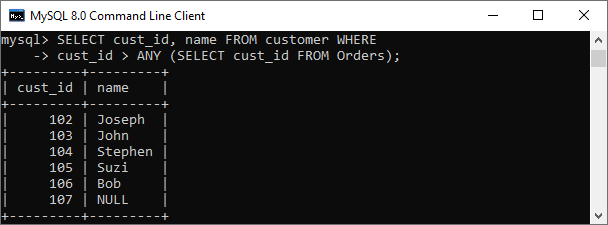
We can use a subquery which is followed by the keyword ALL, ANY, or SOME after a comparison operator. The following are the syntax to use subqueries with ALL, ANY, or SOME:

1. operand comparison\_operator ANY (subquery)
2. operand comparison\_operator ALL (subquery)
3. operand comparison\_operator SOME (subquery)

The ALL keyword compares values with the value returned by a subquery. Therefore, it returns TRUE if the comparison is TRUE for ALL of the values returned by a subquery. The ANY keyword returns TRUE if the comparison is TRUE for ANY of the values returned by a subquery. The ANY and SOME keywords are the same because they are the alias of each other. The following example explains it more clearly:

1. **SELECT** cust\_id, **name** **FROM** customer **WHERE**
2. cust\_id > ANY (**SELECT** cust\_id **FROM** Orders);

We will get the output as follows:



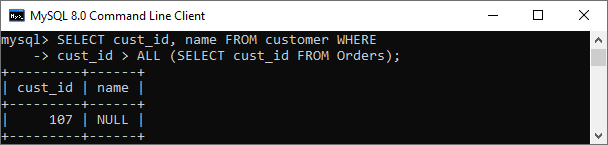
If we use ALL in place of ANY, it will return TRUE when the comparison is TRUE for ALL values in the column returned by a subquery. For example:

Ex: SELECT DEPTNO,DNAME FROM DEPT WHERE DEPTNO>ANY(SELECT DEPTNO FROM EMP);

1. **SELECT** cust\_id, **name** **FROM** customer **WHERE**
2. cust\_id > ALL (**SELECT** cust\_id **FROM** Orders);

EX: SELECT DEPTNO,DNAME FROM DEPT WHERE DEPTNO>ALL(SELECT DEPTNO FROM EMP);

We can see the output as below:



MySQL View

A view is a database object that has no values. Its contents are based on the base table. It contains rows and columns similar to the real table. In MySQL, the View is a **virtual table** created by a query by joining one or more tables. It is operated similarly to the base table but does not contain any data of its own. The View and table have one main difference that the views are definitions built on top of other tables (or views). If any changes occur in the underlying table, the same changes reflected in the View also.

[MySQL](https://www.javatpoint.com/mysql-tutorial)

allows us to create a view in mainly two ways:

1. MySQL Command line client
2. MySQL Workbench

Let us discuss both in detail.

MySQL Command Line Client

We can create a new view by using the **CREATE VIEW** and **SELECT** statement. [SELECT statements](https://www.javatpoint.com/mysql-select)

are used to take data from the source table to make a VIEW.

Syntax

Following is the syntax to create a view in MySQL:

**CREATE** [OR REPLACE] **VIEW** view\_name **AS**

**SELECT** columns

**FROM** tables

[**WHERE** conditions];

Parameters:

The view syntax contains the following parameters:

**OR REPLACE**: It is optional. It is used when a VIEW already exists. If you do not specify this clause and the VIEW already exists, the CREATE VIEW statement will return an error.

**view\_name**: It specifies the name of the VIEW that you want to create in MySQL.

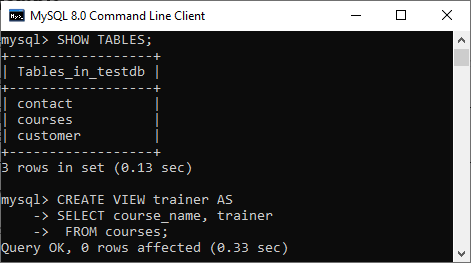
**WHERE conditions**: It is also optional. It specifies the conditions that must be met for the records to be included in the VIEW.

Example

Let us understand it with the help of an example. Suppose our database has a table **course**, and we are going to create a view based on this table. Thus, the below example will create a VIEW name "**trainer**" that creates a virtual table made by taking data from the table courses.

1. **CREATE** **VIEW** trainer **AS**
2. **SELECT** course\_name, trainer
3. **FROM** courses;

Once the execution of the CREATE VIEW statement becomes successful, MySQL will create a view and stores it in the database.



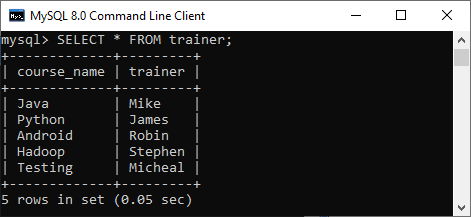
**To see the created VIEW**

We can see the created view by using the following syntax:

**SELECT** \* **FROM** view\_name;

Let's see how it looks the created VIEW:

**SELECT** \* **FROM** trainer;



NOTE: It is essential to know that a view does not store the data physically. When we execute the SELECT statement for the view, MySQL uses the query specified in the view's definition and produces the output. Due to this feature, it is sometimes referred to as a virtual table.

MySQL Update VIEW

In MYSQL, the ALTER VIEW statement is used to modify or update the already created VIEW without dropping it.

**Syntax:**

Following is the syntax used to update the existing view in MySQL:

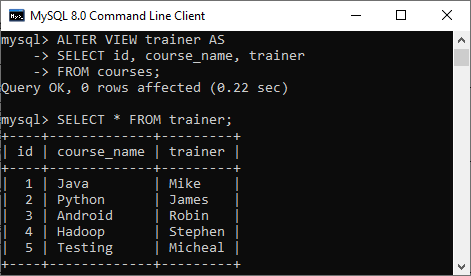
1. **ALTER** **VIEW** view\_name **AS**
2. **SELECT** columns
3. **FROM** **table**
4. **WHERE** conditions;

**Example:**

The following example will alter the already created VIEW name "trainer" by adding a new column.

1. **ALTER** **VIEW** trainer **AS**
2. **SELECT** id, course\_name, trainer
3. **FROM** courses;

Once the execution of the **ALTER VIEW** statement becomes successful, MySQL will update a view and stores it in the database. We can see the altered view using the SELECT statement, as shown in the output:



MySQL Drop VIEW

We can drop the existing VIEW by using the **DROP VIEW** statement.

**Syntax:**

The following is the syntax used to delete the view:

1. **DROP** **VIEW** [IF EXISTS] view\_name;

**Parameters:**

**view\_name**: It specifies the name of the VIEW that we want to drop.

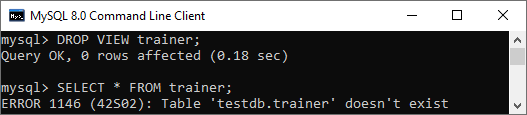
**IF EXISTS**: It is optional. If we do not specify this clause and the VIEW doesn't exist, the DROP VIEW statement will return an error.

**Example:**

Suppose we want to delete the view "**trainer**" that we have created above. Execute the below statement:

1. **DROP** **VIEW** trainer;

After successful execution, it is required to verify the view is available or not as below:

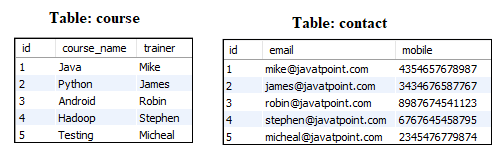


MySQL Create View with JOIN Clause

Here, we will see the complex example of view creation that involves multiple tables and uses a **[join](https://www.javatpoint.com/mysql-join)**

clause.

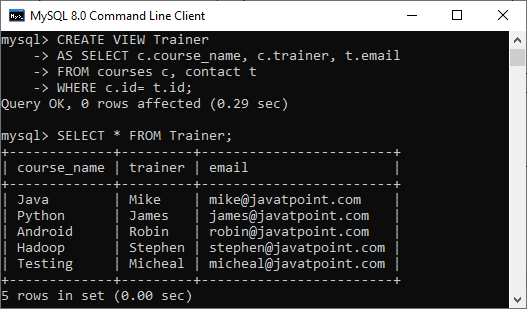
Suppose we have two sample table as shown below:



Now execute the below statement that will create a view Trainer along with the join statement:

1. **CREATE** **VIEW** Trainer
2. **AS** **SELECT** c.course\_name, c.trainer, t.email
3. **FROM** courses c, contact t
4. **WHERE** c.id = t.id;

We can verify the view using the SELECT statement shown in the below image:



## **MySQL CREATE INDEX Statement**

The CREATE INDEX statement is used to create indexes in tables.

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

**Note:** Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched against.

### CREATE INDEX Syntax

Creates an index on a table. Duplicate values are allowed:

CREATE INDEX index\_name  
ON table\_name (column1, column2, ...);

### CREATE UNIQUE INDEX Syntax

Creates a unique index on a table. Duplicate values are not allowed:

CREATE UNIQUE INDEX index\_name  
ON table\_name (column1, column2, ...);

## **MySQL CREATE INDEX Example**

The SQL statement below creates an index named "idx\_lastname" on the "LastName" column in the "Persons" table:

CREATE INDEX idx\_lastname  
ON Persons (LastName);

If you want to create an index on a combination of columns, you can list the column names within the parentheses, separated by commas:

CREATE INDEX idx\_pname  
ON Persons (LastName, FirstName);

## **DROP INDEX Statement**

The DROP INDEX statement is used to delete an index in a table.

ALTER TABLE table\_nameDROP INDEX index\_name;

=================================================================================

CREATE TABLE products (prod\_id INT NOT NULL AUTO\_INCREMENT, prod\_name VARCHAR(20) NOT NULL, prod\_cost FLOAT NOT NULL DEFAULT 0.0, prod\_price FLOAT NOT NULL DEFAULT 0.0, PRIMARY KEY(prod\_id));

INSERT INTO products (prod\_name, prod\_cost, prod\_price) VALUES ('Basic Widget',5.95,8.35),('Micro Widget',0.95,1.35),('Mega Widget',99.95,140.00);

##### **Creating the stored function**

Now that we have a database and a table to work with, we are ready to create a stored function. Let's create a function named **calcProfit**. This function takes two input parameters: the cost and the price of something. It calculates the profit by subtracting the cost from the price, and then returns the value to the calling expression.

To create this stored function, run the following MySQL statements:

DELIMITER $$

CREATE FUNCTION calcProfit(cost FLOAT, price FLOAT) RETURNS DECIMAL(9,2)

BEGIN

DECLARE profit DECIMAL(9,2);

SET profit = price-cost;

RETURN profit;

END$$

DELIMITER ;

*The****DELIMITER****command at the beginning of these statements prevents MySQL from processing the function definition too soon. The****DELIMITER****command at the end of these statements returns processing to normal.*

##### **Using the stored function**

You can now execute the stored function in a database query. The following SQL statement demonstrates how to do this:

SELECT \*, calcProfit(prod\_cost,prod\_price) AS profit FROM products;

This SQL statement returns the following results:

+---------+--------------+-----------+------------+--------+

| prod\_id | prod\_name    | prod\_cost | prod\_price | profit |

+---------+--------------+-----------+------------+--------+

|       1 | Basic Widget |      5.95 |       8.35 |   2.40 |

|       2 | Micro Widget |      0.95 |       1.35 |   0.40 |

|       3 | Mega Widget  |     99.95 |        140 |  40.05 |

+---------+--------------+-----------+------------+--------+

As you can see, the **calcProfit** function automatically calculates the profit (price minus the cost) for each product in the table.

#### Stored procedures

Stored procedures are sometimes confused with stored functions, but they are different in some important ways. Stored procedures, for example, must be invoked with the **CALL** statement, whereas stored functions can be used directly in SQL expressions. You can define and run stored procedures on any A2 Hosting server that uses MySQL.

The following MySQL statements demonstrate how to create a very basic stored procedure named **procedureTest**. This procedure performs a simple lookup on the **products** table that we used in the stored function example above. Although this procedure does not have much practical use, it demonstrates the correct syntax and structure for declaring a stored procedure:

DELIMITER $$

CREATE PROCEDURE procedureTest()

BEGIN

SELECT prod\_name FROM products;

END$$

DELIMITER ;

To execute the stored procedure, use the following MySQL statement:

CALL procedureTest()

DELIMITER $$

CREATE PROCEDURE procedureTest2( in pid int)

BEGIN

SELECT prod\_name FROM products where prod\_id=pid;

END$$

DELIMITER ;

CALL procedureTest(1) ;

DELIMITER $$

CREATE PROCEDURE procedureTest5( out total int)

BEGIN

SELECT count(prod\_name) into total FROM products;

END$$

DELIMITER ;

CALL procedureTest5(@tot) ;

SELECT @tot