CS 338
Lecture 23
Stored Procedures in SQL

Collin Roberts

August 4, 2026

Contents

1. Until now, we have tacitly assumed that the application program which sits on top of our database runs on a different server machine from the database server.
2. This is a good default setup.
3. However, there are situations in which it is useful to create database program modules (procedures or functions) that are stored by the DBMS and executed on the database server.
4. These have historically been called **stored procedures**, although they can be functions or procedures.
5. In the SQL standard, these are called **persistent stored modules (PSM)**, because they are stored persistently in the database, similarly to how data is stored persistently by the DBMS.

1. Text §9.4

Stored procedures are useful in the following circumstances:

1. If the program is needed by several applications, then it can be stored at the server, and invoked by any of the applications programs. This reduces duplication of effort, and improves software modularity.
2. Executing a program at the server can reduce data transfer and communications costs between the client and the server, in certain situations.
3. These procedures can enhance the modelling power provided by **VIEW**s, by allowing more complex types of derived data to be made available to the database users. Additionally, they can be used to check for complex constraints that are beyond the specification power of **ASSERTION**s and **TRIGGER**s.

Stored procedures can also be used to enhance security, as described at the end of this slide deck.

1. Many commercial DBMSs allow stored procedures and functions to be written in a general purpose programming language.
2. Alternatively, a stored procedure can be composed of simple SQL commands such as retrievals an updates.
3. The general form of declaring a stored procedure is as follows:
   CREATE PROCEDURE ⟨ procedure name ⟩ (⟨ parameters ⟩)
   ⟨ local declarations ⟩
   ⟨ procedure body ⟩ ;
4. The parameters and local declarations are optional, and are specified only if needed.

1. For declaring a function, a return type is required, so in this case the declaration form is

CREATE FUNCTION ⟨ function name ⟩ (⟨ parameters ⟩)
RETURNS ⟨ return type ⟩
⟨ local declarations ⟩
⟨ function body ⟩ ;

1. If the procedure (or function) is written in a general purpose programming language, one can specify the language, as well as a file name where the program code is stored.
2. For example, the following format can be used:

CREATE PROCEDURE ⟨ procedure name ⟩ (⟨ parameters ⟩)
LANGUAGE ⟨ programming language name ⟩
EXTERNAL NAME ⟨ file path name ⟩

1. Each parameter should have a **parameter type** that is one of the SQL data types.
2. Each parameter should have a **parameter mode**, which is one of
    1. **IN** (input only),
    2. OUT (output only, i.e. return only) or
    3. INOUT (both input and output).

1. Because procedures and functions are stored persistently by the DBMS, it should be possible to call them from the various SQL interfaces and programming techniques.
2. The CALL statement in the SQL standard can be used to invoke a stored procedure, either from an interactive interface or from embedded SQL.
3. The format of the statement is as follows:

CALL ⟨ procedure or function name ⟩ (⟨ argument list ⟩);

1. SQL/PSM is part of the SQL standard that specifies how to write persistent stored modules.
2. It includes everything we have discussed so far in this slide deck.
3. It also includes additional programming constructs to enhance the power of SQL for the purpose of writing the code (or body) of stored procedures and functions.

4. Here we discuss the SQL/PSM constructs for conditional (branching) statements, and for looping statements.
5. These will give a flavour of the type of constructs that are available.
6. Then we give an example to illustrate how these constructs can be used.

1. The conditional branching statement in SQL/PSM has the following form:

IF ⟨ condition ⟩ **THEN** ⟨ statement list ⟩
ELSEIF ⟨ condition ⟩ **THEN** ⟨ statement list ⟩
ELSEIF ⟨ condition ⟩ **THEN** ⟨ statement list ⟩
ELSE ⟨ statement list ⟩
END IF;

Example

```
   //Function PSM1:
0) CREATE FUNCTION Dept_Size(IN deptno INTEGER)
1) RETURNS VARCHAR [7]
2) DECLARE No_of_emps INTEGER ;
3) SELECT COUNT(*) INTO No_of_emps
4) FROM EMPLOYEE WHERE Dno = deptno ;
5) IF No_of_emps > 100 THEN RETURN 'HUGE'
6)       ELSEIF No_of_emps > 25 THEN RETURN 'LARGE'
7)       ELSEIF No_of_emps > 10 THEN RETURN 'MEDIUM'
8)       ELSE RETURN 'SMALL'
9) END IF ;
```

1. SQL/PSM has several constructs for looping.
2. There are standard while and repeat looping structures, which have the following forms:
   1. While

      WHILE ⟨ condition ⟩ DO
      ⟨ statement list ⟩
      END WHILE;
   2. Repeat

      REPEAT
      ⟨ statement list ⟩
      UNTIL ⟨ condition ⟩
      END REPEAT;

3. There is also a **cursor-based** looping structure.
4. The statement list in such a loop is executed once per tuple in the query result.
5. This has the following form:

FOR ⟨ loop name ⟩ **AS** ⟨ cursor name ⟩ CURSOR FOR ⟨ query ⟩ DO
⟨ statement list ⟩
END FOR;

6. Loops can have names.
7. There is a LEAVE ⟨ loop name ⟩ statement to break a loop when a condition is satisfied.
8. SQL/PSM has many other features, but they are beyond the scope of this slide deck.

1. The fundamental problem that facilitates SQL injection attacks is data being treated as query language.
2. E.g. let query = "SELECT * FROM users WHERE username = '$username' AND password = '$password' ";
3. In this example, if I set $password to "foo' OR 'x'='x' ", then we get this:

   SELECT * **FROM** users **WHERE** username = 'blah'**AND** password = 'foo'**OR** 'x'='x';
4. Since the character 'x' is always equal to the character 'x', this query will always return rows, regardless of whether the user / password combination is correct.

6. The database can't know that you didn't intend this, because it's only being given a string with no context.
7. In order to prevent this, we need to be able to know the difference between the query and the data.
8. Stored procedures solve this problem by writing the query beforehand, with markers for parameters, so that data can be passed into them later.
9. For example:

   SELECT $*$ **FROM** users **WHERE** username $=$? **AND** password $=$?
10. The database driver sends the name of this stored procedure (or, in standard parameterised queries, just the query text itself) and a list of parameters, as distinct separate entities in the protocol.
11. This means that the database server can parse the query string as query language safely, and treat parameters solely as data, without any ambiguity.

1. The theory behind using stored procedures for increased security is that you don't give Data Manipulation Language (DML) access directly to any users.
2. The idea is that users never get anything other than execute permissions on the stored procedures.
3. There is also a slightly less paranoid version of this approach where all writes are by stored procedures, but table/view reads can be done by users.
4. The reasoning goes that if users only have execute permissions on stored procedures and no direct DML access on tables or views, then they can't manipulate data in any unpredictable ways.

5. E.g. let's say you have a bank account system and the only way to make a funds transfer is through a stored procedure.
6. You can have this procedure audit log details about when it happened, where the money went, and so forth.
7. If you allowed direct DML access to the base tables then a compromised account could be used to take money without leaving an audit trail.
8. A compromised account is still a big security hole - big enough to drive a truck through.
9. If all of the compromised account's access is restricted to stored procedure execution however then at least the hole will only fit a smaller truck.

1. Provide an abstraction layer between the application and the database.
2. This enables easier adaptation to changes in the database schema without affecting application code.