New Standard for Stored Procedures in SQL

Andrew Eisenberg
Sybase, Burlington, MA 01803
andrew.eisenberg@sybase.com

1. Introduction

SQL-92/PSM (Persistent Stored Modules) [1] is the second addendum to the SQL-92 ANSI/ISO standard [2] to be approved. This addendum extends SQL-92 with three broad areas of functionality:

- **Multi-statement Procedures:** groups of SQL statements can be executed together; flow-of-control statements, local variables, and condition handlers are provided
- **Stored routines and modules:** procedures, functions, and modules can be stored in an SQL-Server
- **External routines:** functions and procedures written in 3GL host languages can be invoked from SQL statements

In doing this work, the concepts and syntax from modern, block-structured programming languages were used, modified as necessary for the SQL environment.

At the time this article is being written the ISO Editing meeting has just completed successfully, which is the final step in the approval of SQL-92/PSM as an International Standard. This article may contain some omissions or inaccuracies, because the final version of the standard is not yet available.

Throughout this article I will use PSM-96 to refer to SQL-92/PSM. I will include code fragments that do not include error handling, and do not necessarily do useful work.

2. Multi-statement Procedures

SQL-92 allows host languages such as C or COBOL to execute individual SQL statements. To insert a student and an enrollment, a program can be written as follows:

```
void main {
   EXEC SQL INSERT INTO students
      VALUES (10610, 'John Porter', … ) ;
   /* test SQLSTATE for errors */
   EXEC SQL INSERT INTO enrollments
      VALUES (10610, 'CS101', … ) ;
   /* test SQLSTATE for errors */
   }
```

Each of the INSERT statements executes and returns a SQLSTATE value to indicate its success or failure.

2.1 Compound Statements

The compound statement (BEGIN/END) allows these statements to be grouped together.

```
void main {
   EXEC SQL
      BEGIN
         INSERT INTO students
            VALUES (10610, 'Jones', … ) ;
         INSERT INTO enrollments
            VALUES (10610, 'CS101', … ) ;
      END ;
   /* test SQLSTATE for errors */
   }
```

In a client/server environment this can reduce the number of messages that are sent, providing improved performance. Improved performance is likely even in a centralized environment.

2.1.1 Atomicity

In the example above, it is possible that the first insert statement succeeds, and the second insert statement fails due to the failure of a uniqueness constraint. Any partial work of the second insert statement is undone, because SQL considers all DML statements to be atomic. The work of the first statement would not be undone, because the compound statement is a non-atomic statement (as are all of the other flow-of-control statements that will be discussed shortly). PSM-96 provides a second form of the compound statement that is atomic.

```
BEGIN ATOMIC
   INSERT INTO students
      VALUES (10610, 'Jones', … ) ;
   INSERT INTO enrollments
      VALUES (10610, 'CS101', … ) ;
END ;
```

If any statement in the atomic compound statement fails, then the work of all of the statements in the compound statement on the persistent state of the database is undone. Changes that may have been made to variables and parameters are not undone. This action does not end the current transaction.

This is the only statement where an author may choose atomic or non-atomic behavior. The atomic compound statement can be wrapped around any other control statement, routine body, or routine invocation to make its execution atomic.

2.1.2 Variables

The compound statement allows the declaration of variables, which may be of any SQL data type. SQL variables may store a null value or any non-null value of its specified data type. In these ways they differ from host language variables and are similar to the columns of a table.

When a compound statement begins executing, all of the variables that have been declared are created and assigned their initial values. These variables are destroyed when the compound statement ends its execution.

```
BEGIN
   DECLARE start DATE ;

   SELECT   start_date
   INTO     start
   FROM     employees
   WHERE    … ;
END ;
```

2.1.3 Cursors

Compound statements allow the declaration of cursors. As with variables, these local cursors are created when the compound statement begins execution, and destroyed when the execution ends.

```
BEGIN
   DECLARE curs1 CURSOR FOR
      SELECT   name
      FROM     students ;

   OPEN curs1 ;
   FETCH curs1 INTO namecol ;
   CLOSE curs1 ;
END ;
```

2.1.4 Condition Handlers

Every compound statement may specify a set of condition handlers, each of which specify:

- a set of conditions that it is prepared to handle
- an action it will take
- the effect the condition handler will have if the action it has taken is successful

Condition handlers provide the ability to define what actions to take in exceptional circumstances, without writing a test after every statement.

2.1.4.1 Conditions

SQLSTATE values indicate the success or failure of SQL statements. A SQLSTATE value contains a 2 character class value, and a 3 character subclass value.

Class	Description
'00'	Successful completion
'01'	Warning
'02'	Not Found
other	Exception

A condition name may be declared in a compound statement, and may optionally be associated with an SQLSTATE value.

```
BEGIN
   DECLARE enrolled_student CONDITION ;
   DECLARE integrity_violation CONDITION
      FOR SQLSQTATE VALUE '23000' ;
   …
END
```

The conditions that may be specified in a condition handler are:

Condition	Description
SQLSTATE VALUE 'xxyyy'	specific SQLSTATE value
condition name	declared condition
SQLEXCEPTION	SQLSTATE class other than '00', '01', or '02'
SQLWARNING	SQLSTATE class '01'
NOT FOUND	SQLSTATE class '02'

2.1.4.2 Action

The action the handler will take is specified by an SQL statement, which may be a compound statement.

2.1.4.3 Effect

The effect the condition handler has if the action executes successfully is one of the following:

`CONTINUE`	continue execution immediately after the statement that caused the handler to be invoked
`EXIT`	continue execution after the compound statement that contains the handler
`UNDO`	undoes the work of the previous statements in the compound statement, and continues execution after the statement that contains the handler

The UNDO handler may only be specified in a compound statement that specifies ATOMIC.

The example below attempts to insert two rows into the database. If a serialization failure occurs during one of the two insert statements, which can happen due to the activities of concurrent users, then it will retry the statement as many as 3 times before "giving up".

```
BEGIN
   DECLARE i INTEGER DEFAULT 1 ;
   DECLARE success INTEGER DEFAULT 0 ;

   WHILE success = 0 DO
      BEGIN ATOMIC
         DECLARE serialization_failure
            CONDITION FOR SQLSTATE VALUE
            '40001' ;
         DECLARE UNDO HANDLER
            FOR serialization_failure
            BEGIN
               IF i > 3 THEN RESIGNAL ;
               SET i = i + 1 ;
            END ;

         INSERT INTO students … ;
         INSERT INTO housing … ;
         SET success = 1 ;
      END ;
   END WHILE ;
END ;
```

If serialization_failure occurs during one of the insert statements, then the condition handler is executed. Any work completed by the inserts in undone. If i is less than or equal to 3, then the condition handler increments i and execution continues after the atomic compound statement. The WHILE statement will cause the inserts to be retried. If i is greater than 3, then the exception is resignaled. This means that the entire compound statement ends its execution and some condition handler outside of the scope of this example could handle the exception.

2.1.4.4 Unhandled Conditions

If there is no handler for a completion or exception condition that occurs in any lexical scope, then the condition is unhandled.

An unhandled completion condition, such as a warning, will continue execution after the statement that caused the completion condition. An unhandled exception condition will cause the exception condition to be resignaled. If this resignaled condition is not handled by any condition handler in the current execution tree, then the exception condition will eventually be returned to the client application.

Condition handling was covered more extensively in a previous volume of the SIGMOD Record [3].

2.2 Flow-of-Control Statements

PSM-96 provides the flow-of-control statements that one would expect to see in a modern programming language.

The ASSIGNMENT statement assigns a value to a variable or parameter.

The IF statement executes the list of statements in the then clause if the search condition evaluates to TRUE (not FALSE nor UNKNOWN).

```
IF student_state IN ('MA', 'NH')
   THEN stmt; stmt; …
ELSEIF student_country = 'United States'
   THEN stmt; stmt; …
ELSE
   stmt; stmt; …
ENDIF ;
```

The CASE statement has two variations. In one of the variations a single value at the beginning of the CASE statement is checked for equality with a value in each branch of the statement.

```
CASE student_state
WHEN 'MA' THEN stmt; stmt; …
WHEN 'NH' THEN stmt; stmt; …
ELSE stmt; stmt; …
END CASE ;
```

In the other variation each branch of the CASE statement has its own search condition that is evaluated.

```
CASE
WHEN student_state IN ('MA', 'NH')
   THEN stmt; stmt; …
WHEN student_country = 'United States'
   THEN stmt; stmt; …
ELSE stmt; stmt; …
END CASE ;
```

If none of the branches in the CASE statement evaluates to true, then the CASE statement raises an exception.

PSM-96 provides several types of loop statement:

`LOOP`	no test for termination
`WHILE`	tests for termination before executing its statements
`REPEAT`	tests for termination after executing its statements

The WHILE statement is used below to pad a character string with periods.

```
BEGIN
   DECLARE s CHAR VARYING (30) ;

   SET s = ( SELECT   name
             FROM     students
             WHERE    … ) || ' ' ;

   WHILE LENGTH (s) < 30
      SET s = s || '.' ;
   END WHILE ;
END ;
```

A label may be specified at the beginning of a compound statement and at the beginning of many of the flow-of-control statements. The LEAVE statement can then be used to terminate the labeled statement and continue execution with the statement following the labeled statement.

2.2.1 FOR statement

The FOR statement is unlike the statements found in 3GL programming languages. The FOR statement implicitly opens a cursor, fetches the rows of the cursor and executes the body of the FOR statement once for each row, and then closes the cursor.

```
BEGIN
   DECLARE s CHAR VARYING (300)
      DEFAULT '' ;

   FOR x AS SELECT   *
            FROM     enrollments e
            WHERE    e.student = s_id
   DO
      IF s <> ''
      THEN SET s = s || ', ' ;
      END IF ;
      SET s = s || course ;
   END FOR ;
   …
END ;
```

The user is able to specify a name for the cursor in the FOR statement. This cursor name may be used to update or delete the cursor row, but not to close the cursor or change the position of the cursor.

3. Stored Routines

The control statements that have been discussed so far have been discussed in the context of 3GL host language applications, where SQL statements are embedded with the EXEC SQL prefix. The colon is used to identify references to host language variables in the SQL statement.

```
EXEC SQL
   SELECT   nickname
   INTO     :nickname INDICATOR :nn_ind
   FROM     students
   WHERE    … ;
```

These statements are equivalent to an SQL procedure in a client module and a call to the procedure from the host language.

```
{                 /* call in C program */
…
student_nickname
      (&SQLSTATE, nickname, &nn_ind) ;
}

MODULE …          /* SQL client module */
PROCEDURE student_nickname
      (SQLSTATE, nnparm, nnparm_ind) ;
   SELECT   nickname
   INTO     :nnparm INDICATOR :nnparm_ind
   FROM     students
   WHERE    … ;
```

3.1 Stored Procedures

PSM-96 allows procedures to be stored in the database, as objects in a schema, just as tables are. New DDL statements have been added to allow the adding and dropping of these objects.

The mode of the parameters of SQL client procedures were either input, output, both input and output, or none. This mode was inferred from the use of the parameter in the body of the procedure. In PSM-96 the parameter mode may be explicitly specified by the author of the procedure.

Unlike SQL-client procedures, no status parameter needs to be defined. The status of each statement is known to the SQL implementation, and does not need to be explicitly passed from called routine to its caller.

```
EXEC SQL
   CREATE PROCEDURE drop_course
        ( IN student_id CHAR (6),
          IN course_id  CHAR (6),
          OUT transcript_line CHAR (80) )
      BEGIN ATOMIC
         DELETE
         FROM    enrollments
         WHERE   student = student_id
           AND   course = course_id ;

         INSERT INTO transcript VALUES
            ( student_id,
              CURRENT_DATE,
              course_id || ' dropped' ) ;

         SET transcript_line =
                course_id || ' dropped' ;
      END ;

EXEC SQL  /* host language application */
   CALL drop_course ( … ) ;
```

Once a stored procedure has been defined, it can be invoked using the CALL statement. The stored procedure and its parameters are also described in the Information Schema, the read-only views that reflect the database's metadata.

3.2 Stored Functions

SQL-92 supported only procedures. PSM-96 adds functions that can be stored in the database, just as procedures can. Together procedures and functions are referred to as routines.

A function must specify a returns clause, indicating the type of value that it will return. Functions may have only IN parameters, which are the default.

Once a function has been defined, it may be invoked as part of any expression. The RETURN statement ends the execution of the function and provides the value that the function will return. If the function terminates without the execution of a RETURN statement, then an exception is raised.

```
CREATE FUNCTION courses (s_id)
   RETURNS CHAR VARYING (80)

   BEGIN
      DECLARE s CHAR VARYING (80)
         DEFAULT '' ;

      FOR x AS SELECT   *
               FROM     enrollments
               WHERE    e.student = s_id
      DO
         IF s <> ''
            THEN SET s = s || ', ' ;
         END IF ;
         SET s = s || course ;
      END FOR ;

      RETURN s ;
   END ;

SELECT   name, courses (student_id)
FROM     students
WHERE    … ;
```

3.3 Privileges

The list of privileges for SQL objects has been extended to include EXECUTE for routines.

```
GRANT EXECUTE ON drop_course TO admin ;

GRANT EXECUTE ON drop_course TO sr_admin
              WITH GRANT OPTION ;
```

In order to invoke a routine, the user must have been granted execute privilege on the routine.

3.4 Polymorphism and Names

The routines that are created by a user exist within a schema (which exists within a catalog) just as tables and other persistent database objects do. The courses function shown above may be referred to as catalog1.admin.courses.

PSM-96 supports polymorphism. Two routines with the same name can be created in the same schema, as long as the parameters of the two routines are different enough to distinguish them.

3.4.1 Specific Names

In order to distinguish two routines with the same name in the same schema, each routine is also given an alternate and unique name, which is called its specific name. This name can be explicitly specified when the routine is created. If it is not specified, then it will be automatically chosen. A routine can now be identified in several ways in a DROP statement, a GRANT statement, or a REVOKE statement:

```
DROP FUNCTION courses ;

DROP FUNCTION
        courses (CHAR VARYING (80)) ;

GRANT EXECUTE
   ON SPECIFIC FUNCTION courses1000
   TO PUBLIC ;
```

3.4.2 Routine Invocation

With the addition of routines that are polymorphic, the invocation of routines becomes a bit complicated. PSM-96 does not use a distinguished first argument. Instead, we have chosen to consider all of the arguments when selecting which routine to invoke. The former has been referred to as the classical dispatch model and the latter has been referred to as the general dispatch model.

PSM-96 has added a path as an attribute of a schema and a module. The path contains a list of schemas that will be used to locate candidate routines.

PSM-96 defines a precedence among comparable data types, which is used in determining which routine is best for a particular invocation:

```
CHARACTER
   < CHARACTER VARYING

SMALLINT
   < INTEGER
   < DECIMAL
   < …
```

The precedence among the numeric types actually depends on the maximum precision allowed for each numeric type. SMALLINT could have a lower precedence than INTEGER, or it could have the same precedence as INTEGER.

There are several steps that determine which actual routine will be chosen for an invocation.

1. If the invocation is in a call statement, then locate all procedures with the specified name. Otherwise, locate all functions with the specified name.
2. Keep only those routines on which the user has execute privilege.
3. Keep only those routines that have a number of parameters that match the number of arguments. If the parameter is an output parameter, then the data types must match. If the parameter is an input parameter, then the parameter data type must be in the precedence list of the argument data type.
4. If the routine name is qualified, then keep only those routines that are contained in that schema that is specified. If the routine name is unqualified, then keep only those routines that exist in a schema that is included in the path.
5. For each argument, working left to right, do the following:
   a) For all pairs of routines:
      i) If one of the parameters in this position has a higher precedence than the other parameter in this position, remove the routine with the higher precedence.
6. If there is more than one routine left, use the one with the schema that appears earlier in the SQL-path.

3.5 Non-determinism and Side-effects

SQL-92 defines statements and expressions to be either *deterministic*, or *possibly non-deterministic*. Possibly non-deterministic statements and expressions are not allowed in constraint definitions and assertion definitions. A stored routine is possibly non-deterministic if it contains a statement that is possibly non-deterministic.

A stored routine *possibly modifies SQL-data* if it contains a DML statement. Such routines are not allowed in query expressions, constraint or assertion definitions, or search conditions in update or delete statements, because these routines might make the execution of the statements non-deterministic.

```
CREATE FUNCTION add_course ( … )
            RETURNS INTEGER ;
   BEGIN
      INSERT INTO enrollments … ;
      RETURN ( SELECT   COUNT (*)
               FROM     enrollments … ) ;
   END ;

UPDATE  students    -- illegal update stmt
SET     …
WHERE   n_courses = add_course ( … ) ;
```

4. Stored Modules

Client-modules in SQL-92 have the following appearance:

```
MODULE administration
LANGUAGE   C
AUTHORIZATION admin

DECLARE LOCAL TEMPORARY TABLE t1
   ( … ) ;

PROCEDURE hire_teacher ( … ) ;
   BEGIN … END ;
…
```

A form of module may be stored in the database in PSM-96.

```
EXEC SQL
   CREATE MODULE administration

      DECLARE LOCAL TEMPORARY TABLE t1
         … ;

      PROCEDURE hire_teacher (…)
         BEGIN … END ;
      …
   END MODULE ;
```

The module does not require a language clause, because all of the routines it contains can be invoked only by SQL. The authorization identifier for the module is the owner of the schema that contains the module, so this does not need to be specified.

The routines in the module may have SQL bodies, or external bodies. The names used to refer to the routines are the names the routines would have if they were created outside of the module. If the administration module was in the cat1.admin schema, then hire_teacher could be referred to as cat1.admin.hire_teacher.

The drop statement has been extended to allow the module to be dropped. This action drops all of the routines that the module contains. The routines in the module may not be individually dropped.

Similarly, privileges may not be granted to individual routines in the module. Instead the execute privilege may be granted to the module. A user with execute privilege on the module may execute any of the routine that it contains.

A stored module may contain declared local temporary tables, as a client-module does. A stored module may not contain cursor declarations, due to some problems that were discovered just as PSM-96 neared completion.

5. External Routines

PSM-96 allows the use of code written in the familiar host languages to define the body of an SQL routine. The routine has a header part that defines its signature for invocation by SQL statements, just as a stored routine has. Instead of an SQL statement, however, the author provides the information necessary to identify the external code and says something about how parameters will be mapped. Once an external routine has been defined, it is invoked in exactly the same way that SQL-routines are invoked.

The mapping of parameters for external routines is similar to the mapping that occurs when host language code in a client application executes SQL statements. An SQL data type must be mapped to a data type in the host language. An SQL value is possibly the null value, which must be represented in the host language.

An author can specify two ways for this type of mapping to take place.

5.1 PARAMETER STYLE GENERAL

This style of mapping is intended for the use of host language routines that already exist. The host language routine will have one parameter for each parameter in its SQL definition.

```
CREATE FUNCTION soundex
                  (CHAR VARYING (30))
                  RETURNS CHAR (10)
   LANGUAGE C
   EXTERNAL NAME soundex1
   PARAMETER STYLE GENERAL ;

char soundex1 [11]
       (char instring [31]) {
   …
   }

SELECT   *
FROM     students
WHERE    soundex (name)
            = soundex ('Johnston') ;
```

If soundex is invoked with a null value, then an exception will be raised.

5.2 PARAMETER STYLE SQL

This style of parameter mapping is intended for users that wish to write host language code to best support the SQL environment. The return value of the function becomes a host language parameter, after all of the host language parameters that reflect SQL parameters. The SQL parameters and return value each have a host language indicator parameter that can represent the null value.

```
CREATE FUNCTION soundex
                    (CHAR VARYING (30))
                    RETURNS CHAR (10)
   LANGUAGE C
   EXTERNAL NAME soundex2
   PARAMETER STYLE SQL ;

void soundex2
       (char instr [31],
        char outstr [11],
        long *instr_ind,
        long *outstr_ind,
        char exception [6],
        char routine_name [120],
        char specific_name [120],
        char message [81]) {
   …
   }
```

The parameters of soundex2 have the following meanings:

Parameter	Mode	Description
instr	in	input parameter to the SQL function
outstr	out	return value for the SQL function
instr_ind	in	indicator for instr parameter (negative for null value, non-negative otherwise)
outstr_ind	out	indicator for outstr
exception	in/out	SQLSTATE value that indicates success or failure of the routine
routine_name	in	the name of the routine
specific_name	in	the specific name of the routine
message	in/out	message text to be placed in the diagnostics area

PSM-96 is silent on how an external routine is bound to the SQL implementation. PSM-96 is also silent on whether such a routine executes in the same process or address space as the SQL implementation.

5.3 Non-determinism and Side-effects

Because the body of the external routine is written in a host language, PSM-96 cannot infer any of its properties. Instead, the author of the external routine must state whether the routine is deterministic or not deterministic. The author may also state whether the routine possibly modifies SQL-data.

The restrictions on routines that are possibly non-deterministic and possibly modify SQL-data were discussed in section 3.5.

The declarations made by the author concerning determinism cannot be enforced. If the author has declared a routine to be deterministic, then the optimizer is free to cache and reuse results during a transaction. If the author incorrectly states that a routine is DETERMINISTIC, then he or she may get erroneous results.

The declaration that a routine does not modify SQL-data is enforced. An attempt to execute a DML statement in such a routine will raise an exception.

6. Summary

Multi-statement procedures can provide improved performance compared to the execution of individual statements. Stored routines provide improved performance, manageability, and security compared to client procedures. External routines allow an author to use existing host language code, or to write code in a language that may be more suitable to his or her needs.

PSM-96 will be enhanced in the SQL3 standard that is being developed. Features like named parameters and parameter defaults are being considered . SQL3/PSM will serve as the basis for method definition in SQL3 Abstract Data Types (ADT's).

7. References

[1] *ISO/IEC 9075-4:1996, Information Technology - Database Languages - SQL - Part 4: Persistent Stored Modules*, to be published later in 1996. This document will also be known as *ANSI/IEC 9075-4:1996, Information Technology - Database Languages - SQL - Part 4: Persistent Stored Modules.*

[2] *ISO/IEC 9075:1992, Information Technology - Database Languages - SQL, 1992.* This document is also known as *ANS X3.135-1992, Database Language SQL, 1992.*

[3] Richie, J., *"Condition Handling in SQL Persistent Stored Modules,"* SIGMOD Record 24(3), pages 98-103, Sept. 1995.