Condition Handling in SQL Persistent Stored Modules

Jeff Richey

Sybase, Inc.

Introduction

The national and international standards committees responsible for Database Language SQL have proposed a candidate extension for SQL Persistent Stored Modules (SQL/PSM)[1]. The purpose of this extension is to provide a computationally complete language for the declaration and invocation of SQL stored modules and routines. Typically, such routines are stored in a database Server and executed from an application Client in a Client/Server environment.

The proposed SQL/PSM consists of syntax and semantics for variable and cursor declarations, function and procedure (routines) invocations, condition handling, and control statements for looping and branching. An SQL routine is block structured, with each block consisting of local variable and condition handler declarations, a list of SQL statements, and local condition handler execution.

Condition handling is a major new feature of SQL/PSM (henceforth referred to as PSM), although the style and comprehensiveness of the specification is still an issue in further progression of the standard. The specification currently under ballot includes conditions for exceptions, warnings, and other completions such as success of no data, and handlers for Continue, Exit, Redo, and Undo.

Condition handling allows the user to separate condition handling code from the main flow of a routine, thereby eliminating the need to write numerous short and redundant code fragments to handle each unique condition. In some database products, one cannot even resolve the condition in the Server and must instead resort to the Client application program for resolution. Such approaches are often tedious, error-prone, and inflexible. Condition handling in the SQL module avoids these expensive alternatives, instead allowing the procedure to resolve its own conditions and then resume processing.

Condition handling allows one to centralize the handling of conditions and gives users control over two major areas: run-time recovery from failures, and effects of conditions on transactions.

Run-time recovery from failures has the following characteristics:

- allows a user to handle any run-time condition, either by exiting gracefully or by attempting recovery
- provides a recovery mechanism that includes the ability to resolve a condition and then resume action at the statement that caused the condition to be raised (if it was resolved)
- provides the ability to define what "code" will handle each condition

After a condition has been resolved, what is the state of the transaction? Condition handling must ensure that the SQL-data, schemas, and SQL-variables are all maintained in an appropriate *stable* state and can be committed or rolled back. Additionally, the transaction must comply with the ACID test rules[2].

Thus, the benefits of condition handling include:

- allows reduction of error recovery code
- creates a model for trapping and resolving conditions
- provides the ability to resolve the condition and if possible to continue on
- avoids the cost of requiring the SQL-client to resolve the condition
- provides for greater data and path consistency in handling conditions
- separates one condition from another

1.SQL/PSM. ISO/IEC Committee Draft 9075-4, Information Technology - Database Languages - SQL Part 4 - SQL Persistent Stored Modules, document ISO/IEC JTC1/SC21 N8897, August 1994, available from American National Standards Institute, New York, NY.

2.Transaction Process: Concepts and Techniques, Jim Gray and Andreas Reuter, Morgan Kaufman, 1993.

Definitions and Background Information

The following definitions explain some of the terminology used in this paper.

exception condition: an error condition that occurs during execution of an SQL statement. String truncation, invalid cursor state, integrity constraint violation, division by zero, and cardinality violation are all examples of exceptions.

completion condition: a warning or success condition that occurs during execution of an SQL statement. Privilege not granted, null value eliminated in a set function, no data found on a cursor fetch, and successful completion are all examples of completion conditions.

persistent module: an object that consists of routines, cursor declarations, handler and exception declarations, and variable declarations. The object resides within an SQL-server.

routine: a function or procedure that contains SQL procedure statements.

SQL procedure statement: set of SQL statements (DDL, DML, diagnostic, control) that are allowed in a routine.

compound statement: an SQL statement that consists of a list (zero or more) of SQL statements that are enclosed by the keywords BEGIN and END. It can be either atomic or not atomic.

atomic: an SQL procedure statement that either completes successfully or all changes are rolled back. It is the smallest quantum of work.

SQLSTATE value: a character string literal value that represents an exception or completion condition.

exception declaration: a user-specified exception that is referenced by its name.

condition value: an exception name or an SQLSTATE value.

diagnostics area: a place where condition information for an SQL-statement is stored. This area is cleared at the beginning of the execution of an SQL statement unless it is an SQL diagnostics statement.

In the SQL definition, conditions have precedence or "levels". The severity of an error or warning dictates the order of placement in the diagnostics area. The order of precedence is:

- exception conditions that cause SQL-transaction rollback
- exception conditions that cause SQL-statement failure
- completion condition--no data
- completion condition--warning
- completion condition--successful completion

A vendor may add additional conditions such as SQL-server failure, shutdown, resource limits, etc.

Condition Recoverability

What conditions are recoverable? There are many cases where resumption of execution would be undesirable or impossible. For instance, in the case of a system failure, SQL condition handling code will not be able to resolve the condition. In fact, with conditions like these, an SQL session should attempt to exit gracefully.

Depending upon the type of condition, the circumstances leading to the condition, and the desired effects, numerous conditions are recoverable. For example, an attempt to insert a duplicate key value is a recoverable condition. All conditions caused by the application (PSM code in an SQL-server) are generally recoverable.

Condition Handling Models

Most languages/systems do not provide a built-in condition handling model. For the few that support the handling of conditions, that model is either a *termination* or a *resumption* model[3].

The major differences between the two models are that the resumption model provides for more complete user control over condition handling. With the termination model, there are no rules to tell it what to do. It could simply ignore the condition and return an incorrect result or it could "goto" an outer level and thus branch over much of the code that would normally be executed. This can lead to inconsistent behavior for the code and data.

The resumption model provides for greater code path and data consistency. A condition is not the norm, but when it occurs, the behavior should be the same (if the condition can be resolved) as when it does not occur.

---

3. For a limited discussion see: Object-oriented Software Construction, Bertrand Meyer, Prentice Hall, 1988; The Design and Evolution of C++, Bjarne Stroustrup, Addison-Wesley, 1994; Software Engineering with Ada, 2nd Edition, Grady Booch, Benjamin/Cummings, 1987. Contact the author for additional references.

With the resumption model, one can separate out those conditions that they may not be able to resolve. Thus, the model itself is not a limiting factor.

The resumption model is more complex than the termination model. It requires the user to fully understand the appropriate action required to resolve the exception and to continue on executing.

The ideal condition handling model is built on the resumption model and contains the following characteristics:

- gives the user the ability to write condition handling code
- allows for the assigning of one or more SQLSTATE values to the same condition handler code
- allows the actions of the condition handler to be a part of the "nested" transaction
- requires a default condition handler for all exceptions raised for which there are no known associated condition handlers
- specifies behavior for what becomes of actions in atomic and non-atomic statements when a condition is raised
- allows condition handlers to be assigned for all exception, warning, and completion conditions

Condition Handler Declarations

Condition handler declarations (hence forth called *handler declarations*) are SQL statement used to specify the processing of exception and completion condition handling. Below is the bnf for this SQL statement.

DECLARE <handler type> HANDLER
FOR { < SQLSTATE value> | <exception name> }
<handler action>

Handler declarations can be specified in modules and in compound statements. Within the confines of a compound statement, a user may want to undo all of the changes that occurred because a certain condition was raised. This would allow the user to re-stabilize certain SQL-data or schemas that have been modified for that transaction. There are also cases when a user would want to redo the actions within a compound statement (after first undoing them) after they have resolved the condition.

Some situations may arise where a user would want to exit from a compound statement even though that compound statement is only partially complete.

To make condition handling more closely model the information known about a particular block, condition handling must occur at the scope of the handler declaration.

A handler declaration associates one or more condition values with a *handler action*. The *handler action* is either a routine invocation or an SQL procedure statement. Below is the syntax definition.

Condition Handler Types and Definitions

There are four possible *handler types*: CONTINUE, EXIT, REDO, or UNDO.

Continue Handler

When a condition is raised a CONTINUE handler will:

- execute the *handler action*,
- return control to the compound statement from which it was invoked and execute all other SQL-statements following the one that raised the condition.

Below is an example of a CONTINUE handler.

DECLARE CONTINUE HANDLER FOR "01004"
Str_Rtruncation()

When a condition is raised (implicitly or explicitly with the signal statement) control is passed to the code (either an SQL-routine or an SQL procedure statement) specified by the *handler action*. The SQL-statements of the *handler action* are executed. When this work is completed, control is returned to the routine (the compound statement within the routine) that invoked the handler and control is passed to the statement following the one that raised the condition and execution of the compound statement resumes.

CONTINUE handlers are in fact most useful for handling extra processing around completion conditions and those conditions that, for the sake of the current application, are not serious enough to effect the flow of control.

Exit Handler

When a condition is raised an EXIT handler will:

- execute the *handler action*,
- implicitly leave the compound statement at the scope of the handler, with no active condition and does not rollback the compound statement.

Below is an example of an EXIT handler.

DECLARE EXIT HANDLER FOR division_by_zero
Begin Rollback; End

An EXIT handler can be specified in non-ATOMIC compound statements only. Additionally, it gives the effect of the termination model, except that it prevents one from causing a "goto" to an outer block (at any level).

The actions of an EXIT handler could be achieved by coding a LEAVE statement in the *handler action*. However, the EXIT option is a simpler, more readable way to specify this form of action. The technique of using a LEAVE statement can only be used when the handler declaration is specified within the scope of the compound statement in which the condition occurs.

The actions of the EXIT option are similar to the exception handlers of Ada. For those who program with Ada, this is a well understood and widely used style of exception handling.

Redo Handler

A REDO handler causes the compound statement to be undone and then re-executed. When a condition is raised the REDO handler will:

- roll back all of the changes to SQL-data, schemas, or SQL variables by the execution of every SQL-statement contained in the SQL-statement list of the compound statement and cancel any SQL procedure statements triggered by the execution of such statements,
- execute the *handler action*,
- return control to the first SQL-statement of the compound statement at the scope of the handler.

Below is an example of using an REDO handler.

```
DECLARE REDO HANDLER FOR "22008"
        Date_Time_Overflow()
```

When control is returned to the compound statement that raised the exception, it will be re-executed, including the statement that raised the exception. Obviously, if conditional branches in the code existed, then a different path could be taken in the compound statement. This would result in the statement that raised the condition to not be re-executed.

The declaration of a REDO handler can only be specified in a compound statement that is declared to be atomic.

Undo Handler

An UNDO handler causes the compound statement to be undone. When a condition is raised the UNDO handler will:

- roll back all of the changes to SQL-data, schemas, or SQL variables by the execution of every SQL-statement contained in the SQL-statement list of the compound statement and cancel any SQL procedure statements triggered by the execution of any such statements,
- execute the *handler action*,
- return control to the end of the compound statement at the scope of the handler.

Below is an example of using an UNDO handler.

```
DECLARE UNDO HANDLER FOR
    Numeric_Overflow  P_Num_Over()
```

The result is that no SQL-statements in the compound statement are effectively executed.

An undo handler is most useful when the resultant condition is unrecoverable. This will cause all work to be removed and the transaction and data will be left in a stable state.

The declaration of a UNDO handler can only be specified in a compound statement that is declared to be atomic.

Comparison of Condition Handler Types

Below is a table that compares the four different handler types and their characteristics.

Table 1: Comparison of handler types

type	re-sume	roll-back	atom-icity	point of resumption
con-tinue	yes	no	yes	statement following the one that raised the condition
exit	no	no	no	end of compound statement
redo	yes	yes	yes	beginning of compound statement
undo	no	yes	yes	end of compound statement

Condition Handler Action

The *handler action* is either an SQL procedure statement or an SQL-routine. (Remember that an SQL proce-

dure statement can also be a compound statement). Whenever an exception or completion condition is raised from within a routine or an SQL procedure statement, the system looks for an *handler action* for that condition. The attributes of this *handler action* are:

- can be a procedure or SQL statement
- can be declared in either a module or a compound statement
- is one of the following types: continue, exit, redo, or undo
- can contain SQL-statements in the body of the *handler action*
- can itself raise conditions
- provides the mechanism to allow one to either terminate or resume after an exception or completion condition occurs

The execution of the first statement of a *handler action* (that is not a compound statement) deactivates the active exception or completion condition that caused the *handler action* to be invoked, and clears the diagnostics area. If users desire to obtain this information, then they need to use the SQL Get Diagnostics statement as the first statement of the *handler action*.

The *handler action* has access to the following variables or parameters:

- global variables of the module
- parameters and local variables of the routine in which the *handler action* is defined

Additionally, the *handler action* has access to all of the persistent SQL-data or schemas.

If a *handler action* completes with an active exception, then an implicit signal statement is executed. The signal statement determines whether there is another handler declaration in the current module or chain of active modules that can resolve the active condition. If there are none, then control is returned to the SQL-client with this active condition. This type of condition is called an unhandled condition.

Scoping of Condition Handler Declarations

The scope of a handler declaration specified in a compound statement is that compound statement. The scope of a handler declaration specified in a module is that module.

There are possible situations when more than one handler declaration is associated with the same condition value. The handler declaration with the most local scope is chosen as the one associated with that value. One cannot have two or more handler declarations at the same scope that are associated with the same condition value.

Conditions are handled at the scope of the statement raising the condition. Consider the following example:

```
X: BEGIN ATOMIC
    DECLARE UNDO HANDLER FOR
        SQLSTATE VALUE '20002' foo(2);
    Y: BEGIN ATOMIC
        SET G = can-result-in-sqlstate-20002;
    END Y;
END X;
```

When the assignment statement raises the condition, the 'undo' will occur out to the level of the handler declaration. This ensures that block X can always understand the amount of work undone, relative to the work the block itself performs.

The same scoping rules also apply to REDO handlers. When control flow is impacted by a handler, it will be impacted at the scope of the handler declaration.

Signal Statement

The signal statement is used to raise a condition. The syntax for the signal statement is as follows:

```
SIGNAL [ <exception name> | <sqlstate value> ]
        [ ( <argument list> ) ]
```

The signal statement is invoked either implicitly or explicitly.

When a user detects that an event has occurred that is not desirable, they can explicitly cause the *handler action* code associated with that condition to be executed. This occurs by invoking the following statement:

```
SIGNAL Str_RTruncation
```

If the *handler action* code is an SQL-routine that allows arguments, then the following signal statement is executed:

```
SIGNAL Data_Excp_Datetime (a, b, c)
```

where *a*, *b*, and *c* are SQL-variables known to the scope in which the signal statement resides. Normal SQL-routine and argument resolution is employed if the handler declaration associated with an exception name or SQLSTATE value uses an SQL-routine. The effect of this signal statement is to invoke the SQL-routine *HD_C_Excp_Datetime* and pass in as arguments *a*, *b*, and *c*.

If the SQL-server detects that a non-successful condition exists, then it will implicitly cause the *handler action* code for that condition to be executed. This is an

implicit signal statement In effect, the following signal statement is executed:

```
SIGNAL Str_RTruncation
```

The following example illustrates how one could use the signal statement in the resumption model for handling conditions.

```
...
DECLARE Invalid_SP EXCEPTION;
...
PROCEDURE Add_SP (
    IN      Sno             CHARACTER(30),
    IN      Pno             CHARACTER(30),
    IN      Qty             DECIMAL(5),
    IN      Origin          CHARACTER(30))
BEGIN ATOMIC
    IF Sno = '' OR Pno = '' THEN
        SIGNAL Invalid_SP (Sno, Pno);
    END IF
    INSERT INTO SP VALUES(Sno, Pno, Qty,
    Origin);
END
```

The above example may be able to resolve the empty string values for the SQL-variables *Sno* and *Pno*. For example, the SQL-routine *Invalid_SP()* may call out to some external routine and obtain proper values for these SQL-variables. This would allow *Add_SP()* to be re-executed.

Argument Passing and Handler Declarations

For an implicit signal statement, one needs to be able to specify the arguments to be passed in when the execution of that statement occurs. If the condition is raised at the same level as where the handler declaration is specified, then all is fine. Otherwise, one may want to override the arguments specified in the handler declarations.

Suppose that a handler declaration was specified for string truncation at the module level (because we want to make it available for use throughout the entire module). At an inner level, one determines that the local variable which contains the string is different from the one specified at the module level; and if a condition is raised (at this level), then we want to use this local variable. Below is an example.

```
MODULE Supplier_Parts
...
DECLARE str CHAR(50);
...
DECLARE CONTINUE HANDLER
    FOR SQLSTATE VALUES '22001'
    R_Str_RTrunc ( str, len, act_len );
...
PROCEDURE Add_SP ( IN len_r INT, IN a_len_r
    INT )
BEGIN ATOMIC
    DECLARE str_r  CHAR(40);
    ...
    A: BEGIN ATOMIC
        DECLARE CONTINUE HANDLER
            FOR SQLSTATE VALUES '22001'
            R_Str_RTrunc ( str_r, len_r, a_len_r );
        ...
    END A;
END
```

Conclusion

Condition handling is a critical and significant feature in SQL-92 PSM. It gives users control over run-time recovery from failures and effects of conditions on transactions.

Additionally, condition handling allows users to separate condition handling code from the main flow of a routine. Thus, the routine can resolve its own conditions and resume processing without leaving the Server and avoiding the expense of Client alternatives which are tedious, error-prone, and inflexible.

Further, using the resumption model as the basis for condition handling gives to users the capability to define the action which is appropriate for their given conditions.

The author wishes to thank Amelia Carlson, Len Gallagher, Nelson Mattos, Janet Prichard, and Phil Shaw for their valuable contributions. Due to the requirements for brevity, this article does not contain full and complete examples.

The author's e-mail address is: jeff.richey@sybase.com