Rdb Journal
technical corner

SQL Procedural Language

In Rdb Release 7.1

A feature of Oracle Rdb

By Ian Smith
Oracle Rdb Relational Technology Group
Oracle Corporation

The examples in this article use SQL language from Oracle Rdb V7.1 and later versions.

SQL Procedural Language in Rdb 7.1

This is the second article in a series on the SQL Procedural Language. In the first part I listed the language features available in Oracle Rdb7 and discussed the benefits in packaging complex application logic and executing it on the Rdb Server.

We will now examine the procedural language enhancements made to Oracle Rdb V7.1. The general theme was to allow more modular design and construction of applications. To this end global variables can be shared between stored procedures and functions, routines can use optional arguments, the DEFAULT keyword can be passed so that the argument adapts to changes in the database environment, tables can be locked using LOCK TABLE, and so on.

Local Variables

In previous versions the DECLARE statement allocated local variables for use by compound statements. Their scope was limit to that of the begin-end block that declared them.

Rdb 7.1 now eliminates any local variable that was not used in the application. Interactive SQL, the SQL module language and the SQL precompiler report an informational message. These variables should be removed from the compound statement to improve the readability of the code.

Although interactive SQL supports a subset of this syntax, it is really only used to declare host variables, such as those defined in a C or COBOL application. These types of variables do not support NULL assignment unless an accompanying indicator variable.

Module Global Variables

Rdb now supports global variables that can be reference by all routines in the module that declares them. These variables can be given initial values and they will hold persistent data across call statements.

Contrast this with local variables that are initialized each time the compound statement is entered. Note that variables without DEFAULT clauses will have an undefined value.

In prior releases global data could be associated with a module using a declared temporary table (also known as a scratch table). The following examples show the two techniques.

```
SQL> create module SAMPLE_70
cont>     language SQL
cont>     comment is 'use scratch table to hold persistent data'
cont>
cont>     declare local temporary table module.counter (val
cont>          integer) on commit preserve rows
cont>
cont>     procedure SHOW_COUNTER;
cont>     trace 'Current Value: ', (select val from
cont>          module.counter);
cont>
cont>     function GET_COUNTER
cont>     returns integer;
cont>     return (select val from module.counter);
cont>
cont>     procedure ADD_COUNTER (in :incr integer);
cont>     case (select count (*) from module.counter)
cont>     when 0 then insert into module.counter values (:incr);
cont>     else update module.counter set val = val + :incr;
cont>     end case;
cont>
cont>     procedure INIT_COUNTER;
cont>     delete from module.counter;
cont>     end module;
```

The Rdb 7.1 version is much more efficient since it uses direct assignment to the variable and does not involve the overhead of the INSERT, UPDATE or SELECT statements.

```
SQL> create module SAMPLE_71
cont>     language SQL
cont>     comment is 'use global variables to hold
cont>          persistent data'
```

```
cont>
cont>     declare :counter integer = NULL
cont>
cont>     procedure SHOW_COUNTER;
cont>     trace 'Current Value: ', COALESCE
cont>          (:counter, '** NULL **');
cont>
cont>     function GET_COUNTER
cont>     returns integer;cont>    return :counter;
cont>
cont>     procedure ADD_COUNTER (in :incr integer);
cont>     set :counter = COALESCE (:counter, 0) + :incr;
cont>
cont>     procedure INIT_COUNTER;
cont>     set :counter = NULL;
cont> end module;
```

The initial value for the variable is assigned just prior to the execution of the first procedure or function in the module. This DEFAULT value can be quite complex using subqueries, arithmetic expressions and calling stored or external functions.

Note: the functions referenced by the DEFAULT must exist prior to the execution of the CREATE MODULE statement and therefore cannot reference routines within this module.

Optional Routine Parameters

As applications evolve and business rules change there is a need within the SQL programming environment to revise the parameters to stored functions and procedures. However, adding a new parameter may require searching for callers in SQL module language sources, as well as other stored procedures.

To make maintenance of complex applications easier Rdb now allows trailing IN parameters of routines to be omitted within compound statements. Rdb supplies missing parameters using the parameter DEFAULT value or NULL if there was no DEFAULT used.

```
SQL> create module EMPLOYEE_UTILS
cont>     procedure SHOW_EMPLOYEE
```

```
cont>          (in :eid id_dom default '00000'
cont>               comment is 'search employee id',
cont>           in :max_rows integer default 2000
cont>               comment is 'limit to rows displayed');
cont>     begin …body… end;
cont> end module;
```

As Rdb fills in the optional parameters the called routine does not need to do any special processing of the argument to handle the truncated parameter list. In this example the follow CALL statements would be permitted:

- Call SHOW_EMPLOYEE ();
- Call SHOW_EMPLOYEE ('00164');
- Call SHOW_EMPLOYEE ('00245', :max_li);

The use of DEFAULT and optional parameters is permitted for both external and SQL stored routines. This support applies only to function calls and to the CALL statement in a compound statement.

Note: a COMMENT IS clause is also allowed so that the routine parameters can be fully described.

DEFAULT function

A new function is provided for the INSERT value list, UPDATE set statement, CALL statement and function parameter lists. This function returns the default value defined for the column or parameter.

The programmer can force the currently define DEFAULT to be assigned or passed to a routine and so avoid embedding these values in the application.

The following two calls are equivalent to omitting the trailing parameters in out previous example:

- Call SHOW_EMPLOYEE (DEFAULT, DEFAULT);
- Call SHOW_EMPLOYEE ('00164', DEFAULT);

The DEFAULT can be used to force defaulting for other parameters that cannot be omitted.

- Call SHOW_EMPLOYEE (DEFAULT, :max_li);

Note: DEFAULT is applied in the context of the caller, so a default of CURRENT_USER will return the user of the caller and not that of the routine. This is an important distinction for definers rights stored procedures.

For INSERT and UPDATE the DEFAULT applies the columns default value, or NULL if the column is not assigned a default.

```
SQL> create table PERSON
cont>     (f_name char(30),
cont>      start_date timestamp default current_timestamp);
SQL> insert into PERSON values (default, default)
```

This INSERT statement is equivalent to the new syntax in Rdb 7.1

```
SQL> insert into PERSON default values;
```

The advantage of this statement is that it tolerates changes to the table. If new columns are added then it will still assign the default to all columns.

DECLARE statement

With Rdb V7.1.0.2 a new CHECK clause can be used for local variables. This check clause is evaluated each time a value is assigned to the variable. The SET statement, CALL statement, and the RETURNING clauses of UPDATE and INSERT perform assignment.

If the CHECK constraint is applied to more than one variable then you must use the VALUE keyword to indirectly reference the current variable. If the check constraint fails then an exception is raised. Application programmers can user this mechanism to enforce value ranges.

Case Statement

Rdb enhanced the CASE statement allowing lists of value expressions, and a new searched CASE statement.

The simple case statement allowed a single literal value to be used in the WHEN clause, this format has been enhanced to allow a list of literal values to be used, making it a more compact alternative to the IF THEN ELSEIF statement.

The following program fragment shows the new format being used to process a range of values.

```
CASE :selected_char
     WHEN '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
          THEN SET :char_type = 0  -- number
     WHEN '_', '$'
          THEN SET :char_type = 1  -- punctuation
     ELSE SET :char_type = 2;      -- other
END CASE;
```

The searched format of the CASE statement closely resembles the CASE expression, allowing complex Boolean conditions for each WHEN clause. Application programmers can now choose between CASE and the IF-THEN-ELSE statement that provide similar functionality.

REPEAT Statement

In prior releases Rdb supported a single LOOP statement which could be modified by the WHILE clause to limit the number of iterations. The SQL:1999 language standard has altered the syntax slightly so that the WHILE statement must be terminated by END WHILE instead of the previous END LOOP clause. Rdb V7.1 supports both variations.

To these two statements is added a REPEAT statement. Some care should be taken with this statement since the loop termination condition does not accept NULL as exit criteria and any UNTIL clause that evaluates to NULL may cause an infinite loop.

```
SQL> begin
cont> declare :c integer = 100;
cont>     repeat
cont>          set :c = :c – 1;
cont>          trace ‘iteration ‘, :c;
cont>     until :c < 0 or :c is null
cont>     end repeat;
cont> end;
```

Counted FOR loop

Rdb 7.1 introduces a new type of FOR loop that iterates through a defined range of values. This iterator must be explicitly declared as a local variable, global variable or as a routine parameter.

```
SQL> begin
cont>     declare :lc integer;
cont>     for :lc in 1 to (select count (*) from departments)
cont>     do
cont>          trace :lc;
cont>     end for;
cont> end;
```

Within the FOR counted loop the local variable is considered read-only and may not be updated using the SET statement, CALL statement, or the RETURNING clauses of UPDATE and INSERT.

A REVERSE keyword can be used decrement through the range. The default step size is 1. In Rdb V7.1.0.2 a new STEP clause was added to allow arbitrary increment sizes and the data types permitted for the iterator variable now include INTERVAL types of a single field.

Note: the new CHECK constraint support for variable is implicitly used to ensure that the STEP clause is a positive integer value.

LEAVE Statement

The LEAVE statement in prior versions required that the statement, or loop be labeled so that the LEAVE statement could use this label. The one exception was that the procedure name could be used as an implicit label to exit the stored procedure.

In this release the label is now optional. If omitted the LEAVE statement will exit the inner most loop, or if there is none then the next labeled statement, and finally if no labeled statement is found it will leave the procedure. This relaxed syntax allows easier coding and more readable procedures.

Note: you must use the RETURN statement to leave a stored function, as a result is required. Attempts to leave without executing a return statement will cause an exception to be raised.

ITERATE Statement

The ITERATE statement was added to the SQL standard for Persistent Stored Modules in SQL:1999. It is similar to the LEAVE statement as it changes the current flow of the procedure. It causes the loop to restart on the next iteration. This is a welcome addition to the procedural statements as complex loops can now be simplified; eliminating extra labels, local flags and IF or CASE conditionals.

As with the LEAVE statement the label is optional.

SIGNAL Statement

The SIGNAL statement was introduced in Rdb 7.0. This statement allows the programmer to abort a procedure and return a system or user defined SQLSTATE to the application.

With this release of Rdb the syntax fully conforms to SQL:1999 and is also extended to permit user defined text to be returned.

```
SQL> begin
cont>     ...statements...
cont>     if :sc > 0 then
cont>         signal sqlstate 'RR000' ('PAYROLL Table is empty');
cont>     end if;
cont> end;
%RDB-E-SIGNAL_SQLSTATE, routine "(unnamed)" signaled SQLSTATE
"RR000"
-RDB-I-TEXT, PAYROLL Table is empty
```

The returned string is optional and can be any value expression, so that the returned message makes sense to the caller. If the SQLSTATE is one that is defined by Rdb it will be mapped to an existing SQL error message. If the code is not known then the generic message ***SIGNAL_SQLSTATE*** will be used and the optional text will be visible to the application.

LOCK TABLE Statement

A transaction can be started using two methods; one is a generic SET TRANSACTION that allows any tables to be accessed during the transaction; and the other is using the SET TRANSACTION … RESERVING clause to specify the initial locking modes for several tables. The latter method enforces the list of tables that can be accessed by the application. The exceptions are; Rdb system tables, tables that are referenced by constraints, trigger actions and COMPUTED BY columns (7.0.6.3 and 7.1.0.1 and later versions). These are automatically included in the permitted tables list.

The RESERVING clause is an important tool as it permits the table to be initially locked in a higher mode (such as PROTECTED WRITE) and eliminates the lock transaction from the default of SHARED READ. This lock transition is often a cause of lock conflicts or deadlocks because several applications may access a table for shared read and then encounter a lock conflict when trying to upgrade the lock for the update action. If the transaction explicitly reserves the table at the higher mode then access will be serialized and no lock conflict will be encountered.

The RESERVING clause is vital in many applications but it poses a problem for the SQL programmer when databases and applications evolve. These SET TRANSACTION statements must be updated to include the new tables used by stored procedures and functions otherwise those new routines will fail.

To accommodate this type of application change Rdb has added the LOCK TABLE statement that is closely related to the RESERVING clause of the SET TRANSACTION statement. It adds the specified table to the current list of permitted tables. The SQL programmer would include this in their new stored procedures to ensure that the table can be accessed.

The following example shows an attempt to reference JOB_HISTORY when the SET TRANSACTION statement only specified the EMPLOYEES table. A subsequent LOCK TABLE allows access.

```
SQL> set transaction reserving employees for shared read;
SQL> select count(*) from job_history where employee_id = '00164';
%RDB-E-UNRES_REL, relation JOB_HISTORY in specified request is not
a relation reserved in specified transaction
SQL> lock table job_history for shared read mode;
SQL> select count(*) from job_history where employee_id = '00164';
          2
1 row selected
```

As with other reserved tables the table locks are released on the next COMMIT or ROLLBACK statement. The lock modes may only be escalated, e.g. you cannot reduce the locking mode from EXCLUSIVE to SHARED using this statement.

START TRANSACTION Statement

This statement is part of SQL:1999 and acts in Rdb just like the SET TRANSACTION statement. Currently, the SQL:1999 standard only supports the clauses READ ONLY, READ WRITE, and ISOLATION LEVEL.

START DEFAULT TRANSACTION Statement

When a PROFILE is created for a user it may contain a DECLARE TRANSACTION clause that describes the default transaction characteristics for the user. The START DEFAULT TRANSACTION statement will use those characteristics at runtime, so it is sensitive to the current users profile.

This has interesting implications for applications. The following example starts a default transaction and then tests the characteristics to see if the user should be updating data.

```
SQL> set flags 'trace';
SQL> begin
cont> declare :tt rdb$object_name;
cont>
cont> start default transaction;
cont>
cont> get diagnostics :tt = TRANSACTION_TYPE;
cont> trace 'Transaction type: ', :tt;
cont>
cont> if :tt = 'READ ONLY' then
cont>     call REPORT_TABLE ();
cont> else
cont>     call UPDATE_TABLE ();
cont> end if;
cont>
cont> commit;
cont>
cont> end;
~T Compile transaction (6) on db: 1
~T Transaction Parameter Block: (len=0)
~Xt: Transaction type: READ ONLY
~Xt: read table
```

Please refer to the CREATE PROFILE statement for more details on default transactions.

COMMIT AND CHAIN
ROLLBACK AND CHAIN

Rdb supports a new AND CHAIN syntax supported for COMMIT and ROLLBACK. When AND CHAIN is used a new transaction is implicitly started using the same attributes as the previously committed or rolled back transaction. Attributes such as READ WRITE, READ ONLY, RESERVING, EVALUATING, WAIT, and ISOLATION LEVEL are retained for the new transaction.

Applications can use this new clause to simplify applications, since the complex transaction attributes need only be specified once.

The AND CHAIN clause is only permitted in a compound statement (i.e. in a BEGIN ... END block), or as the body of a stored procedure.

When the SET FLAGS option TRANSACTION_PARAMETERS is specified a line of output is written to identify the chained transaction. Each SET TRANSACTION assigns a unique sequence number that is displayed after each transaction action line.

```
~T Restart_transaction (3) on db: 1, db count=1
```

This output shows that a transaction has been restarted and includes the transaction execution id, the database attach sequence id, and the number of databases involved in this transaction.

The following example executes SET TRANSACTION once at the start of the procedure. Then periodically the transaction is committed and restarted using the COMMIT AND CHAIN syntax. This simplifies the application since there is only one definition of the transaction characteristics.

```
SQL> -- process table in batches
SQL>
SQL> set compound transactions 'internal';
SQL> set flags 'transaction,trace';
SQL>
SQL> begin
cont> declare :counter integer = 0;
cont> declare :emp integer;
cont>
```

```
cont> set transaction
cont>     read write
cont>     reserving employees for exclusive write;
cont>
cont> for :emp in 0 to 600
cont> do
cont>     begin
cont>     declare :id char(5)
cont>           default substring (cast (:emp+100000 as
cont>                           varchar(6)) from 2 for 5);
cont>     if exists (select * from employees where
cont>                           employee_id = :id)
cont>     then
cont>         trace 'found: ', :id;
cont>         if :counter > 20
cont>         then
cont>             commit and chain;
cont>             set :counter = 1;
cont>         else
cont>             set :counter = :counter + 1;
cont>         end if;
cont>     end if;
cont>     end;
cont> end for;
cont>
cont> commit;
cont> end;
~T Compile transaction (1) on db: 1
~T Transaction Parameter Block: (len=2)
0000 (00000) TPB$K_VERSION = 1
0001 (00001) TPB$K_WRITE (read write)
~T Start_transaction (1) on db: 1, db count=1
~T Rollback_transaction on db: 1
~T Compile transaction (3) on db: 1
~T Transaction Parameter Block: (len=14)
0001 (00001) TPB$K_WRITE (read write)
0002 (00002) TPB$K_LOCK_WRITE (reserving) "EMPLOYEES"
TPB$K_EXCLUSIVE
```

```
~T Start_transaction (3) on db: 1, db count=1
~Xt: found: 00164
~Xt: found: 00165
.
.
.
~Xt: found: 00185
~T Commit_transaction on db: 1
~T Prepare_transaction on db: 1
~T Restart_transaction (3) on db: 1, db count=1
~Xt: found: 00186
~Xt: found: 00187
.
.
.
~Xt: found: 00435
~Xt: found: 00471
~T Commit_transaction on db: 1
~T Prepare_transaction on db: 1
SQL>
```

References:

Information and complete syntax for these new features can be found in these documents:

- Oracle Rdb New and Changed Features for Oracle Rdb
  Release 7.1 for OpenVMS Alpha
  Part Number A90804-01

- Oracle® Rdb
  Release Notes
  Release 7.1.0 for OpenVMS Alpha
  June 2001

- Oracle® Rdb for OpenVMS
  Release Notes
  Release 7.1.0.1
  November 2001

- Oracle® Rdb for OpenVMS
  Release Notes
  Release 7.1.0.2
  April 2002

Oracle Rdb
Procedural Language Enhancements for Rdb Release 7.1
May 2002

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com