Panther

Database Driver–MS SQL Server

Release 4.25

Prolifics®

May 2000

1

Database Driver for SQL Server

This document is designed as a supplement to information found in the *Developer's Guide*.

Initializing the Database Engine

Database engine initialization occurs in the source file `dbiinit.c`. This source file is unique for each database engine and is constructed from the settings in the `makevars` file. In Prolifics for SQL Server, this results in the following `vendor_list` structure in `dbiinit.c`:

```
static vendor_t vendor_list[] =
{
   {"sqlsrvr", dm_msssup, DM_DEFAULT_CASE ,(char *) 0},

   { (char *) 0, (int (*)()) 0, (int) 0, (char *) 0 }
};
```

The settings are as follows:

`sqlsrvr`	Engine name. May be changed.
`dm_msssup`	Support routine name. Do not change.
`DM_DEFAULT_CASE`	Case setting for matching `SELECT` columns with Prolifics variable names. May be changed.

For Prolifics for SQL Server, the settings can be changed by editing the `makevars.mss` file.

Engine Name

You can change the engine name associated with the support routine `dm_msssup`. The application then uses that name in `DBMS ENGINE` statements and in `WITH ENGINE` clauses. For example, if you wish to use "tracking" as the engine name, change the following parameter in the `makevars.mss` file:

```
MSS_ENGNAME=tracking
```

If the application is accessing multiple engines, it makes SQL Server the default engine by executing:

where *sqlsrvr-engine-name* is the string used in `vendor_list`. For example,

```
DBMS ENGINE sqlsrvr
```

or

```
DBMS ENGINE tracking
```

Support Routine Name

`dm_sup` is the name of the support routine for SQL Server. This name should not be changed.

Case Flag

The case flag, `DM_DEFAULT_CASE`, determines how Prolifics's database drivers use case when searching for Prolifics variables for holding `SELECT` results. This setting is used when comparing SQL Server column names to either a Prolifics variable name or to a column name in a `DBMS ALIAS` statement.

SQL Server is case-sensitive. SQL Server uses the exact case of a SQL statement when creating database objects like tables and columns. In subsequent SQL statements, you must use the same exact case when referring to these objects. The default setting for case-sensitive engines is `DM_PRESERVE_CASE`. This means that the SQL Server column name is matched to a Prolifics variable with the same name and case when processing `SELECT` results.

The case setting can be changed. You can force Prolifics to perform case-insensitive searches. Substitute the `l` option in the `makevars` file to match SQL Server column names to lower case Prolifics variables, or use the `u` option to match to upper case Prolifics variables.

```
MSS_INIT=l
```

or

```
MSS_INIT=u
```

If you edit `makevars.mss`, you must remake your Prolifics executables. For more information on engine initialization, refer to Chapter 7 in the *Developer's Guide*.

Connecting to the Database Engine

SQL Server allows your application to use one or more connections. The application can declare any number of named connections with `DBMS DECLARE CONNECTION` statements, up to the maximum number permitted by the server.

The following options are supported for connections to SQL Server:

Table 1. Database connection options.

Option	Argument
`USER`	*user-name*
`INTERFACES`	*interfaces-file-pathname*
`SERVER`	*server-name*
`DATABASE`	*database-name*
`PASSWORD`	*password*
`APPLICATION`	*application-name*
`CHARSET`	*character-set-name*
`CURSORS`	*1 | 2*
`TIMEOUT`	*seconds*
`HOST`	*host-name*
`SQLTIMEOUT`	*seconds*

```
DBMS [ WITH ENGINE engine ] DECLARE connection CONNECTION \
   [ FOR [ USER user-name ] [ PASSWORD password ] \
   [ DATABASE database ] [ SERVER server ] \
   [ APPLICATION application-name ] [ CURSORS number-of-cursors ] \
   [ HOST host-name ] [ INTERFACES interface-file-pathname ] \
   [ SQLTIMEOUT seconds ] [ TIMEOUT seconds ] [ CHARSET character-set ] ]
```

For example:

```
DBMS DECLARE dbi_session CONNECTION FOR \
   USER ":uname" PASSWORD ":pword" DATABASE "sales" \
   SERVER "sybase10" APPLICATION "sales" HOST "oak" \
   INTERFACES "/usr/sybase/interfaces.app" \
   CURSORS "2" SQLTIMEOUT "120" TIMEOUT "15"
```

Additional keywords are available for other database engines. If those keywords are included in your `DBMS DECLARE CONNECTION` command for SQL Server, it is treated as an error.

Importing Database Tables

The Import⇒Database Objects option in the screen editor creates Prolifics repository entries based on database tables in an SQL Server database. When the import process is complete, each selected database table has a corresponding repository entry screen.

In Prolifics for SQL Server, the following database objects can be imported as repository entries:

- database tables
- database views

After the import process is complete, the repository entry screen contains:

- A widget for each column in the table, using the column's characteristics to assign the appropriate widget properties.
- A label for each column based on the column name.
- A table view named for the database table or database table view.
- Links that describe the relationship between table views.

Each import session allows you to display and select up to 1000 database tables. Each database table can have up to 255 columns. If your database contains more than 1000 tables, use the filter to control which database tables are displayed.

Table Views

A table view is a group of associated widgets on an application screen. As a general rule, the members of a table view are derived from the same database table. When a database table is first imported to a Prolifics repository, the new repository screen has one table view that is named after the database table. All the widgets corresponding to the database columns are members of that table view.

The import process inserts values in the following table view properties:

- Name — The name of the table view, generally the same as the database table.
- Table — The name of the database table.
- Primary Keys — The columns that are defined as primary keys or unique indexes for the database table.
- Columns — A list of the columns in the database table is displayed when you click on the More button. However, this list is for reference only. It cannot be edited.
- Updatable — A setting that determines if the data in the table can be modified. The default setting for Updatable is Yes.

For each repository entry based on a database view, the primary key widgets must be available if you want to update data in that view. To do this, check that the Prolifics table view's Primary Keys property is set to the correct value. Then, the widgets corresponding to the primary keys must be members of either the Prolifics table view or one of its parent table views. For repository entries based on database tables, this information is automatically imported.

Links

Links are created from the foreign key definitions entered in the database. The application screen must contain links if you are using the transaction manager and the screen contains more than one table view.

Check the link properties to see if they need to be edited for your application screen. The Parent and Child properties might need to be reversed or the Link Type might need to be changed.

Refer to Chapter 30 in the *Developer's Guide* for more information on links.

Widgets

A widget is created for each database column. The name of the widget corresponds to the database column name. The Inherit From property is set to `@DATABASE` indicating that the widget was imported from the database engine. The Justification property is set to Left. Other widget properties are assigned based on the data type.

The following table lists the values for the C Type, Length, and Precision properties assigned to each SQL Server data type.

SQL Server Data Type	Code	Prolifics Type	C Type	Widget Length	Widget Precision
`binary`	45	`DT_BINARY`	`Hex Dec`	column length * 2	
`bit`	50	`FT_INT`	`Int`	1	
`char`	47	`FT_CHAR`	`Char String`	column length	
`datetime`	61	`DT_DATETIME`	`Default`	17	
`decimal`	55				
scale > 0		`FT_FLOAT`	`Float`	column precision + column scale + 1	column scale
else		`FT_LONG`	`Long Int`	column precision	

SQL Server Data Type	Code	Prolifics Type	C Type	Widget Length	Widget Precision
`double precision`	62	`FT_FLOAT`	`Float`	16	2
`float`	62	`FT_FLOAT`	`Float`	16	2
`image`	34	`DT_BINARY`	`Hex Dec`	column length	
`int`	56	`FT_LONG`	`Long Int`	11	
`money`	60	`DT_CURRENCY`	`Default`	26	
`nchar`	47	`FT_CHAR`	`Char String`	column length	
`nvarchar`	47	`FT_CHAR`	`Char String`	column length	
`numeric`	63				
scale > 0		`FT_FLOAT`	`Float`	column precision + column scale + 1	column scale
else		`FT_LONG`	`Long Int`	column precision	
`real`	59	`FT_FLOAT`	`Float`	16	2
`smalldatetime`	58	`DT_DATETIME`	`Default`	17	
`smallint`	52	`FT_INT`	`Int`	6	
`smallmoney`	122	`DT_CURRENCY`	`Default`	14	
`text`	35	`FT_CHAR`	`Char String`	254	
`timestamp`	80	`DT_BINARY`	`Hex Dec`	column length	
`tinyint`	48	`FT_INT`	`Int`	3	
`varbinary`	37	`DT_BINARY`	`Hex Dec`	column length * 2	
`varchar`	39	`FT_CHAR`	`Char String`	column length	

Other Widget Properties

Based on the column’s data type or on the Prolifics type assigned during the import process, other widget properties might be automatically set when importing database tables.

UseInUpdate property

If a column’s length is defined as larger than 254 in the database, then the database importer sets the Use In Update property to No for the widget corresponding to that column. Because widgets in Prolifics have a maximum length of 254, the data originally in the database column could be truncated as part of a `SAVE` command in the transaction manager.

The Use In Update property is also set to No for certain data types. In SQL Server, this applies to the data types `text`, `image`, and for any `numeric` column that is defined as `identity`.

DT_CURRENCY `DT_CURRENCY` widgets have the Format/Display⇒Data Formatting property set to Numeric and Format Type set to 2 Dec Places.

DT_DATETIME `DT_DATETIME` widgets also have the Format/Display⇒Data Formatting property set to Date/Time and Format Type set to `DEFAULT`. Note that dates in this Format Type appear as:

```
MM/DD/YY HH:MM
```

Null Field property If a column is defined to be `NOT NULL`, the Null Field property is set to No. For example, the `roles` table in the `videobiz` database contains three columns: `title_id`, `actor_id` and `role`. `title_id` and `actor_id` are defined as `NOT NULL` so the Null Field property is set to No. `role`, without a `NOT NULL` setting, is implicitly considered to allow null values so the Null Field property is set to Yes.

For more information about usage of Prolifics type and C type, refer to Chapter 29 of the *Developer's Guide*.

Formatting for Colon Plus Processing and Binding

This section contains information about the special data formatting that is performed for the engine. For general information on data formatting, refer to Chapter 29 in the *Developer's Guide*.

Formatting Dates

Prolifics uses SQL Server's `convert` function and the SQL Server format string, `yyyymmdd hh:mm:ss` to convert a Prolifics date-time format to a SQL Server format.

In order for conversion to take place, the widget must have the C Type set to Default and the Format/Display⇒Data Formatting property set to Date/Time. Any date-time Format Type is appropriate.

This is the format for literal dates. It is compatible with SQL Server national language support.

Formatting Currency Values

SQL Server requires a leading dollar sign for values inserted in a `money` column in order to ensure precision. Prolifics will use a leading dollar sign when it formats widgets with a Prolifics type of `DT_CURRENCY`. Any other amount formatting characters are stripped. Therefore, if a currency field contained

```
500,000.00
```

Prolifics would format it as

```
$500000.00
```

Using Text and Image Data Types

Note that when the select list includes the values of text and image data types, the limit on the length of the data returned depends on the server setting of `textsize`. The SQL Server server default is 32K; however, this value can be changed on the server via the SQL Server `set` command. The global variable `@@textsize` contains the current maximum.

Declaring Cursors

Each Prolifics cursor uses a SQL Server `dbprocess`. By default, Prolifics for SQL Server uses one cursor (`dbprocess`) for operations performed by `DBMS SQL`. Therefore, if an application executes the sequence:

```
DBMS SQL SELECT ...
DBMS SQL UPDATE ...
```

the following command to display additional rows in the select set:

```
DBMS CONTINUE
```

will fail because SQL Server discards the select set when the cursor is re-used.

Prolifics for SQL Server supports a connection option of `CURSORS 2` for simulating two default cursors. When this option is used, Prolifics for SQL Server opens two default cursors on each connection. It uses one cursor for all `SELECT` statements. It uses the second cursor for all non-`SELECT` statements; this includes `INSERT`, `UPDATE`, `DELETE`, and all stored procedure calls. Transaction commands (`BEGIN`, `COMMIT`, `ROLLBACK`) are also issued for the non-`SELECT` cursor.

If you use the `CURSORS 2` connection option, you will need to declare a named cursor to execute a stored procedure (or SQL batch command) that returns select rows. The second default cursor never returns select rows.

Prolifics does not put any limit on the number of cursors an application may declare to an SQL Server engine. Because each cursor requires memory and SQL Server resources, however, it is recommended that applications close a cursor when it is no longer needed.

For more information on cursors, refer to Chapter 27 in the *Developer's Guide*.

Scrolling

SQL Server has native support for non-sequential scrolling in a select set. This capability is available on any cursor. As an alternative, you can switch to Prolifics scrolling. Both systems allow you to use the following commands:

`DBMS` *[* `WITH CURSOR` *cursor-name]* `CONTINUE_BOTTOM`

`DBMS` *[* `WITH CURSOR` *cursor-name]* `CONTINUE_TOP`

`DBMS` *[* `WITH CURSOR` *cursor-name]* `CONTINUE_UP`

For native scrolling, use this command:

`DBMS` *[* `WITH CURSOR` *cursor-name]* `SET_BUFFER` *number-of-rows*

This command sets the DB-Library option `DBBUFFER`. When this command is used, SQL Server buffers the specified number of select rows in the program's memory.

For Prolifics scrolling, use this command::

`DBMS` *[* `WITH CURSOR` *cursor-name]* `STORE FILE` *[filename]*

To turn off Prolifics scrolling and close the continuation file, use this command:

`DBMS` *[* `WITH CURSOR` *cursor-name]* `STORE`

or close the Prolifics cursor with `DBMS CLOSE CURSOR`.

For more information on scrolling, refer to Chapter 28 in the *Developer's Guide*.

Locking Behavior

Prolifics developers using SQL Server should consider locking issues when building applications that select large amounts of data.

When an application executes a SQL SELECT that returns many rows, SQL Server might use a “shared lock” on each data page to preserve read-consistency. That is, to preserve the state of the selected data, SQL Server might prevent other applications or users from changing the data until the application has received all the rows. This behavior is usually seen for select sets that contain several hundred rows.

As a part of developing and testing an application, you should monitor SQL Server’s behavior by running the SQL Server command sp_lock from another terminal when the application executes a SELECT. If a SELECT executed by a Prolifics application is holding a lock, the cursor’s *spid* will be listed.

Because a shared lock prevents other users from updating data, it is important to release shared locks as soon as possible. To release a shared locked, you must either:

- Get all the rows in the select set.
- Flush pending rows in the select set.

An application has two ways of getting the entire select set:

- Create Prolifics arrays that are large enough to hold the entire select set.
- Use DBMS STORE FILE and DBMS CONTINUE_BOTTOM to buffer all the rows in a temporary file on disk.

For example, an application might set up a continuation file before executing a SELECT. Before returning control to the user, the application might execute DBMS CONTINUE_BOTTOM, which forces Prolifics get all the rows from the select set and buffer them in a temporary file. This also forces SQL Server to release any shared lock it is holding for the SELECT.

In the following example, the application puts a message on the status line and flushes the display. Next it sets up a continuation file and executes the SELECT. It calls DBMS CONTINUE_BOTTOM to force Prolifics to get all the rows. Finally, it calls DBMS CONTINUE_TOP to ensure that the select set’s first page (rather than its last page) of rows is displayed when control is returned to the user.

```
proc big_select
   msg setbkstat ”Processing. Please be patient...”
   flush
   DBMS STORE FILE
   DBMS SQL SELECT ....
   DBMS CONTINUE_BOTTOM
   DBMS CONTINUE_TOP
   msg d_msg ” ”
return
```

An application can also limit the number of rows a user can view at a time by using the `DBMS FLUSH` command. When this command is executed, SQL Server discards any pending rows and releases all associated locks. For example,

```
proc big_select
   DBMS SQL SELECT ....
   if @dmretcode != DM_NO_MORE_ROWS
      DBMS FLUSH
return
```

To monitor lock information within the application, the application can query SQL Server for the `spid` (server process id) number of a cursor and the number of locks held by the cursor. Note that each cursor has its own `spid` and it keeps the same `spid` number until the application closes the cursor. To get a cursor's `spid` number, an application must use the cursor to select the global SQL Server variable `@@spid`.

```
# Get the SQL Server spid for a Prolifics cursor
# before SELECTing rows.
proc get_spid (cursor)
vars spid
   if cursor == ""
      DBMS SQL SELECT spid = @@spid
   else
   {
      DBMS DECLARE :cursor CURSOR FOR \
         SELECT spid = @@spid
      DBMS EXECUTE :cursor
   }
   return spid

# Get the number of locks held by a SQL Server spid.
proc lockstatus (spid4select)
   vars lcount
   DBMS DECLARE lock_cursor CURSOR FOR \
      SELECT COUNT(*) FROM master.dbo.syslocks \
      WHERE spid = :spid4select
   DBMS WITH CURSOR lock_cursor ALIAS lcount
   DBMS WITH CURSOR lock_cursor EXECUTE
   DBMS CLOSE CURSOR lock_cursor
   return lcount
```

An application can get a cursor's spid before executing a `SELECT` for rows. After fetching rows the application can query SQL Server for the number of locks. Note that the order of these statements is important: if an application attempts to get a cursor's spid *after* fetching rows, the `SELECT` for the cursor's spid will release any locks and any pending rows. For this reason, be sure to get the cursor's spid *before* fetching rows. Refer to the example below.

```
proc select
vars cursor_spid, locks_before, locks_after

   cursor_spid = get_spid (”c1”)
   locks_before = lockstatus (cursor_spid)

   DBMS DECLARE c1 CURSOR FOR SELECT ...
   DBMS WITH CURSOR c1 EXECUTE

   locks_after = lockstatus (cursor_spid)
   if locks_after > locks_before
      msg emsg ”The SELECT has locked data.”

return 0
```

Error and Status Information

Prolifics uses the global variables described in the following sections to supply error and status information in an application. Note that some global variables can not be used in the current release; however, these variables are reserved for use in other engines and for use in future releases of Prolifics for SQL Server.

Errors

Prolifics initializes the following global variables for error code information:

`@dmretcode`	Standard database driver status code.
`@dmretmsg`	Standard database driver status message.
`@dmengerrcode`	SQL Server error code.
`@dmengerrmsg`	SQL Server error message.
`@dmengreturn`	Return code from an executed stored procedure.

SQL Server returns error codes and messages when it aborts a command. It usually aborts a command because the application used an invalid option or because the user did not have the authority required for an operation. Prolifics writes SQL Server error codes to the global variable `@dmengerrcode` and writes SQL Server messages to `@dmengerrmsg`.

All SQL Server errors with a severity greater than 10 are Prolifics errors. Otherwise, they are considered warnings.

Using the Default Error Handler

The default error handler displays a dialog box if there is an error. The first line indicates whether the error came from the database driver or database engine, followed by the text of the statement that failed. If the error comes from the database driver, `Database interface` appears in the `Reported by` list along with the database engine. The error number and message contain the values of `@dmretcode` and `@dmretmsg`. If the error comes from the database engine, only the name of the engine appears in the `Reported by` list. The error number and message contain the values of `@dmengerrcode` and `@dmengerrmsg`.

Using an Installed Error Handler

An installed error or exit handler should test for errors from the database driver and from the database engine. For example:

```
DBMS ONERROR JPL errors
DBMS DECLARE dbi_session CONNECTION FOR ...

proc errors (stmt, engine, flag)
if @dmengerrcode == 0
   msg emsg ″JAM error: ″ @dmretmsg
else
   msg emsg ″JAM error: ″ @dmretmsg ″ %N″ \
   ″:engine error is ″ @dmengerrcode ″ ″ @dmengerrmsg
return 1
```

For additional information about engine errors, refer to your SQL Server documentation. For more information about error processing in Prolifics, refer to Chapter 36 in the *Developer's Guide* and Chapter 12 in the *Programming Guide*.

Warnings

Prolifics initializes the following global variables for warning information:

`@dmengwarncode`	SQL Server warning code.
`@dmengwarnmsg`	SQL Server warning message.

Prolifics writes the code to `@dmengwarncode` and the message to `@dmeng-warnmsg`.

A warning usually describes some non-fatal change in the SQL Server environment. For example, SQL Server issues a warning when the application changes a connection's default database.

You might wish to use an exit hook function to process warnings. An exit hook function is installed with `DBMS ONEXIT`. A sample exit hook function is shown below.

```
proc check_status (stmt, engine, flag)

if @dmengwarncode
   msg emsg ":engine Warning is " @dmengwarnmsg
return
```

Row Information

Prolifics initializes the following global variables for row information:

`@dmrowcount`	Count of the number of SQL Server rows affected by an operation.
`@dmserial`	Not used in Prolifics for SQL Server.

SQL Server returns a count of the rows affected by an operation. Prolifics writes this value to the global variable `@dmrowcount`.

As explained on the manual page for `@dmrowcount`, the value of `@dmrowcount` after a SQL `SELECT` is the number of rows fetched to Prolifics variables. This number is less than or equal to the total number of rows in the select set. The value of `@dmrowcount` after a SQL `INSERT`, `UPDATE`, or `DELETE` is the total number of rows affected by the operation. Note that this variable is reset when another `DBMS` statement is executed, including `DBMS COMMIT`.

The value of `@dmrowcount` might be unexpected after executing a stored procedure. This is documented SQL Server behavior. If you need this information, SQL Server recommends that you test for it within the stored procedure and return it as an output parameter or return code. `@@rowcount` is a SQL Server global variable. For example:

```
create proc update_ship_fee @class int, @change float
as
declare @u_count int
update cost set ship_fee = ship_fee * @change
   where class = @class
select @u_count = @@rowcount
return @u_count
```

Refer to your SQL Server Command Reference Manual for more information.

Using Stored Procedures

A stored procedure is a precompiled set of SQL statements that are recorded in the database and executed by calling the procedure name. Since the SQL parsing and syntax checking for a stored procedure are performed when the procedure is created, executing a stored procedure is faster than executing the same group of SQL statements individually. By passing parameters to and from the stored procedure, the same procedure can be used with different values. In addition to SQL statements, stored procedures can also contain control flow language, such as `if` statements, which gives greater control over the processing of the statements.

Database engines implement stored procedures very differently. If you are porting your application from one database engine to another, you need to be aware of the differences in the engine implementation.

Executing Stored Procedures

An application can execute a stored procedure with `DBMS SQL` and the engine's command for execution, `EXEC`. For example:

```
DBMS SQL [ DECLARE parameter data-type \
    [ DECLARE parameter data-type ... ] ] \
    EXEC procedure-name [ parameter [ OUT ][, parameter [ OUT ]...] ]
```

An application can also use a named cursor to execute a stored procedure:

```
DBMS DECLARE cursor CURSOR FOR \
    [ DECLARE parameter data-type [ DECLARE parameter data-type ... ] ] \
    EXEC procedure-name [ parameter [ OUT ][ , parameter [ OUT ]...] ]
```

The cursor can then be executed with the following statement:

```
DBMS [ WITH CURSOR cursor ] EXECUTE [ USING values ]
```

Example

For example, `update_tapes` is a stored procedure that changes the video tape status to `O` whenever a video is rented.

```
create proc update_tapes @parm1 int, @parm2 int
as
update tapes set status = 'O'
   where title_id = @parm1 and copy_num = @parm2
```

The following statement executes this stored procedure, updating the `status` column of the `tapes` table using the onscreen values of the widgets `title_id` and `copy_num`.

```
DBMS SQL EXEC update_tapes :+title_id, :+copy_num
```

A `DECLARE CURSOR` statement can also execute a stored procedure. First, a cursor is declared identifying the parameters. Then, the cursor is executed with a `USING` clause that gets the onscreen values of the widgets `title_id` and `copy_num`.

```
DBMS DECLARE x CURSOR FOR EXEC update_tapes \
   ::parm1, ::parm2
DBMS WITH CURSOR x EXECUTE USING title_id, copy_num
```

Remember to use double colons (::) in a `DECLARE CURSOR` statement for cursor parameters. If a single colon or colon-plus were used, the data would be supplied when the cursor was declared, not when it was executed. Refer to Chapter NO TAG in the *Developer's Guide* for more information.

Getting Output Parameter Values

If the DBMS supports output parameters, the keyword `OUT` traps the value of an output parameter in a Prolifics variable. For example, the stored procedure `rent_summary` calculates the total number of rentals for the day and the total price paid for those rentals.

```
create proc rent_summary
   @num_rented int output, @tot_price output, @day datetime
as
create table rentsum (price money)
insert into rentsum select rentals.price from rentals
  where rental_date = @day
select @num_rented = count(*) from rentsum
select @tot_price = sum (price) from rentsum
drop table rentsum
```

The application should declare a cursor for the procedure:

```
DBMS DECLARE cur1 CURSOR FOR \
   declare @t1 int declare @t2 money \
   EXEC rent_summary @num_rented=@t1 OUT, \
   @tot_price=@t2 OUT, @day =::today
DBMS WITH CURSOR cur1 EXECUTE USING today = day
```

Note that `t1` and `t2` are temporary SQL Server variables, not Prolifics variables. SQL Server requires that output values be passed as variables, not as constants. If `num_rented` and `tot_price` are Prolifics variables, the procedure returns the number of videos rented on a specific day and the total price paid for those videos. The application can use `DBMS ALIAS` to map the values of output parameters to Prolifics variables. You can modify the previous procedure so that it maps the value of of `num_rented` to the Prolifics variable `vid_count` and the value of `tot_price` to the Prolifics variable `total_paid`:

```
DBMS DECLARE cur1 CURSOR FOR \
   declare @t1 int declare @t2 money \
   EXEC rent_summary @num_rented=@t1 OUT, \
   @tot_price=@t2 OUT, @day =::today
DBMS WITH CURSOR cur1 ALIAS num_rented vid_count, \
   tot_price total_paid
DBMS WITH CURSOR cur1 EXECUTE USING today = day
```

Using Remote Procedure Calls

In addition to the `EXEC` command, SQL Server supports a remote procedure call ("rpc") for executing a stored procedure. You should consider using rpc rather than `EXEC` when either the following occur:

- One or more of the stored procedure's parameters has a data type that is not `char`. An rpc is more efficient in these cases because it is capable of passing parameters in their native data types rather than only as ASCII characters. This reduces the amount of data conversion for the application and the server.
- The stored procedure returns output parameters. An rpc provides a faster and simpler mechanism for accommodating output parameters.

To make a remote procedure call, an application performs the following steps:

- Must declare an rpc cursor.
- Must declare the data type of each parameter that has a non-char data type.
- May specify aliases for output parameters or selected columns.
- Must execute the cursor, supplying in the `USING` clause a Prolifics variable for each parameter.

The sections below describe these steps in detail. Examples follow.

Declaring the rpc Cursor

Prolifics uses binding to support rpc's. Therefore, to execute a stored procedure with an rpc, the application must declare an rpc cursor. The syntax is the following:

```
DBMS [WITH CONNECTION connection] \
   DECLARE cursor CURSOR FOR RPC procedure \
   [ ::parameter [OUT] [, ::parameter [OUT]...] ]
```

The keyword `RPC` is required. Following the keyword is the name of the procedure and the names of the procedure's parameters. All parameters must begin with a double colon, the Prolifics syntax for cursor parameters. The name of the bind parameter must be the same parameter name used in the procedure. If a parameter is an output parameter, the keyword `OUT` should follow the parameter name if the application is to receive its value.

Datatyping the rpc Parameters

To pass parameters in their native data types, the application must specify a data type for each non-character parameter. The syntax for `DBMS TYPE` is the following:

`DBMS` *[* `WITH CURSOR` *cursor]* `TYPE` *[parameter] engine-data-type* `\`
[, [parameter] engine-data-type ...]

parameter is a parameter in the `DECLARE CURSOR` statement. *engine-data-type* is the data type of a parameter in the procedure. If parameter names are not given, the types are assigned by position.

Prolifics uses the information in the `DBMS TYPE` statement to make the required calls to add parameters to an rpc. Please note that `DBMS TYPE` has no effect on the data formatting that is performed for binding.

Redirecting the Value of Output Parameter

By default, when an rpc cursor with an output parameter is executed, a search is performed for a Prolifics variable with the same name as the output parameter. To write the output value to a Prolifics variable with another name, use the `DBMS ALIAS` command.

`DBMS` *[* `WITH CURSOR` *cursor]* `ALIAS` *[output_parameter] variable* `\`
[, [output_parameter] variable ...]

If the procedure selects rows, aliases can be given for the tables' columns. If the procedure returns output parameters and column values, aliases should be given by name rather than by position.

Executing the rpc Cursor

The application executes the stored procedure by executing the rpc cursor. The `USING` clause must provide a Prolifics variable for each parameter. The syntax is the following:

`DBMS` *[* `WITH CURSOR` *cursor]* `EXECUTE \`
`USING` *[parameter =] variable [, [parameter =] variable ...]*

Prolifics passes the name of the parameter given in the `DBMS DECLARE CURSOR` statement, the data type of the parameter given in the `DBMS TYPE` statement, and the parameter's value which is the value of *variable*.

Parameters and Prolifics variables can be bound either by name or by position. The two forms should not be mixed, however, in one statement.

Example

`cust_rent` calculates the new total `rent_amount` column in the `customers` table.

```
CREATE PROC cust_rent
   @cid int, @crent money, @rprice money,
   @newrent money output
AS
SELECT @crent = (select rent_amount from customers
   where cust_id = @cid)
SELECT @newrent = @crent + @rprice
```

An rpc is more efficient than an exec cursor because the procedure has an input parameter with a non-character data type, and because it returns an output parameter.

The following statement declares an rpc cursor for the stored procedure. The names of the bind parameters match the parameters in the stored procedure. Note that the keyword `OUT` follows the output parameter.

```
DBMS DECLARE cur2 CURSOR FOR RPC cust_rent ::cid, ::crent, \
   ::rprice, ::newrent OUT
```

Before executing the cursor, the application must specify the SQL Server data types for any non-character data types.

```
DBMS WITH CURSOR cur2 TYPE \
   cid int, crent money, rprice money, newrent money
```

When executing the cursor, the application must provide a Prolifics variable for each parameter. Prolifics passes the name, data type, and value of the parameters to the procedure. Note that the procedure does not use the input value of the parameter `newrent`. Prolifics's binding mechanism, however, requires a variable in the `USING` clause for each parameter.

```
DBMS WITH CURSOR cur2 EXECUTE cust_rent \
   USING cust_id, rent_amount, price, newrent
```

The procedure passes its output, the new total, to the Prolifics variable `newrent`.

If instead, you wish to put the output value in the widget `rent1`, execute the following:

```
DBMS WITH CURSOR cur2 ALIAS newrent rent1
DBMS WITH CURSOR cur2 EXECUTE cust_rent USING cid=cust_id, \
   crent=rent_amount, rprice=price, newrent=rent1
```

Note that the variable names in the `USING` clause do not affect the destination of output values when the cursor is executed. Only a `DBMS ALIAS` statement can remap the output variables to other Prolifics variables.

Of course, this procedure can also be executed with the standard `EXEC` cursor. It would require the following declaration,

```
DBMS DECLARE cur3 CURSOR FOR \
   declare @x money \
   EXEC cust_rent @cid = ::cust_id, @crent = ::rent_amount, \
   @rprice = ::price, @newrent = @x output

DBMS WITH CURSOR cur3 EXECUTE cust_rent \
   USING cid=cust_id, crent=rent_amount, rprice=price, \
   newrent=newrent
```

Getting a Return Code from a Stored Procedure

Prolifics provides the global variable `@dmengreturn` to trap the return status code of a stored procedure. This variable is empty unless a stored procedure explicitly sets it. Note that the variable will not be set until the procedure has completed execution. Therefore, an application should evaluate the value of `@dmengreturn` when `@dmretcode = DM_END_OF_PROC`.

Executing a new `DBMS` statement clears the value of `@dmengreturn`.

If `multiply` is the following stored procedure,

```
create proc multiply @m1 int, @m2 int,
   @guess int output, @result int output
as
select @result = @m1 * @m2
if @result = @guess
   return 1
else
   return 2
```

the application should set up variables for the output parameters.

Either an rpc cursor or an exec cursor can be declared and executed for the procedure that calculates the values in the Prolifics variables `m1` and `m2` and then writes the values of the output parameters `guess` and `result` to the Prolifics variables `attempt` and `answer`.

```
# RPC cursor
DBMS DECLARE x CURSOR FOR \
   RPC multiply ::m1, ::m2, ::guess OUT, ::result OUT
DBMS WITH CURSOR x TYPE m1 int, m2 int, \
   guess int, result int
DBMS WITH CURSOR x ALIAS guess attempt, result answer
DBMS WITH CURSOR x EXECUTE USING m1, m2, attempt, answer

# EXEC cursor
DBMS DECLARE y CURSOR FOR \
   declare @syb_tmp1 int \
   declare @syb_tmp2 int \
   select @syb_tmp1 = ::user_guess\
   EXEC multiply @m1=::p1, @m2=::p2, \
      @guess= @syb_tmp1 OUT, @result= @syb_tmp2 OUT
DBMS WITH CURSOR y ALIAS guess attempt, result answer
DBMS WITH CURSOR y EXECUTE \
   USING user_guess = attempt, p1 = m1, p2 = m2
```

After executing the cursor, the application can test the value of `@dmengreturn` and display a message based on the return status code.

```
proc check_ret
if @dmretcode == DM_END_OF_PROC
{
   if @dmengreturn == 1
      msg emsg ”Good job!”
   else if @dmengreturn == 2
      msg emsg ”Better luck next time.”
}
else
{
   DBMS NEXT
   call check_ret
}
return
```

Controlling the Execution of a Stored Procedure

Prolifics’s database driver for SQL Server provides a command for controlling the execution of a stored procedure that contains more than one `SELECT` statement. The command is:

`DBMS` *[* `WITH CURSOR` *cursor]* `SET` *behavior*

behavior can have one of these values:

`STOP_AT_FETCH`

`EXECUTE_ALL`

If *behavior* is `STOP_AT_FETCH`, Prolifics stops each time it executes a non-scalar `SELECT` statement in the stored procedure. Therefore, a `SELECT` from a table will halt the execution of the procedure. However, a `SELECT` of a single scalar value (i.e., using the SQL functions `SUM`, `COUNT`, `AVG`, `MAX`. or `MIN`) does not halt the execution of a stored procedure.

The application can execute

`DBMS` *[* `WITH CURSOR` *cursor]* `CONTINUE`

or any of the `CONTINUE` variants to scroll through the selected records. To abort the fetching of any remaining rows in the select set, the application can execute

`DBMS` *[* `WITH CURSOR` *cursor]* `FLUSH`

To execute the next statement in the procedure the application must execute

`DBMS` *[* `WITH CURSOR` *cursor]* `NEXT`

`DBMS NEXT` automatically flushes any pending `SELECT` rows.

To abort the execution of any remaining statements in the stored procedure or the `sql` statement, the application can execute

`DBMS [WITH CURSOR` *cursor* `] CANCEL`

All pending statements are aborted. Canceling the procedure also returns the procedure's return status code. The return code `DM_END_OF_PROC` signals the end of the stored procedure.

If *behavior* is `EXECUTE_ALL`, Prolifics executes all statements in the stored procedure without halting. If the procedure selects rows, Prolifics returns as many rows as can be held by the destination variables and continues executing the procedure. The application cannot use the `DBMS CONTINUE` commands to scroll through the procedure's select sets.

Note that SQL Server does not support `SINGLE_STEP` as an option for stored procedure execution; however, it is available for execution of multi-statement cursors.

Using Transactions

A transaction is a unit of work that must be totally completed or not completed at all. SQL Server has one transaction for each cursor. Therefore, in a Prolifics application, a transaction controls all statements executed with a single named cursor or the default cursor.

Applications that need transaction control on multiple cursors should use two-phase commit service.

The following events commit a transaction on SQL Server:

- Executing `DBMS COMMIT`.
- Executing a data definition command such as `CREATE`, `DROP`, `RENAME`, or `ALTER`.

The following events roll back a transaction on SQL Server:

- Executing `DBMS ROLLBACK`.
- Closing the transaction's cursor or connection before the transaction is committed.

Note that SQL Server will not rollback remote procedure calls (rpcs) or data definition commands that create or drop database objects. Refer to the SQL Server documentation for more information on these restrictions.

Transaction Control on a Single Cursor

After an application declares a connection, an application can begin a transaction on the default cursor or on any declared cursor.

SQL Server supports the following transaction commands:

- Begin a transaction on a default or named cursor.

  ```
  DBMS [WITH CONNECTION connection] BEGIN
  DBMS [WITH CONNECTION cursor] BEGIN
  ```

- Commit the transaction on a default or named cursor.

  ```
  DBMS [WITH CONNECTION connection] COMMIT
  DBMS [WITH CONNECTION cursor] COMMIT
  ```

- Rollback to a savepoint or to the beginning of the transaction on a default or named cursor.

  ```
  DBMS [WITH CONNECTION connection] ROLLBACK [savepoint]
  DBMS [WITH CONNECTION cursor] ROLLBACK [savepoint]
  ```

- Create a savepoint in the transaction on a default or named cursor.

  ```
  DBMS [WITH CONNECTION connection] SAVE [savepoint]
  DBMS [WITH CONNECTION cursor] SAVE [savepoint]
  ```

A transaction on a default cursor controls all inserts, updates, and deletes executed with the JPL command `DBMS SQL`. The application can set the default connection before beginning the transaction or it can use the `WITH CONNECTION` clause in each statement.

If a named cursor is declared for multiple statements, it might be useful to execute the cursor in a transaction. This way, the application can ensure that SQL Server executes either all of the cursor's statements or none of the cursor's statements. A simple transaction on a named cursor might appear like this:

```
DBMS DECLARE cursor CURSOR FOR statement [statement...]
DBMS WITH CURSOR cursor BEGIN
DBMS WITH CURSOR cursor EXECUTE [USING parm [parm ...]]
...
DBMS WITH CURSOR cursor COMMIT
```

If necessary, the cursor can be executed more than once in the transaction. The application should not, however, redeclare a cursor within a transaction.

Example

The following example contains a transaction on the default connection with an error handler.

```
# Call the transaction handler and pass it the name
# of the subroutine containing the transaction commands.

call tran_handle ”new_title()”

proc tran_handle (subroutine)
{
# Declare a variable jpl_retcode and
# set it to call the subroutine.
   vars jpl_retcode
   jpl_retcode = :subroutine

# Check the value of jpl_retcode. If it is 0, all statements
# in the subroutine executed successfully and the transaction
# was committed. If it is 1, the error handler aborted the
# subroutine. If it is -1, Prolifics aborted the subroutine.
# Execute a ROLLBACK for all non-zero return codes.

   if jpl_retcode == 0
   {
      msg emsg ”Transaction succeeded.”
   }
   else
   {
      msg emsg ”Aborting transaction.”
      DBMS ROLLBACK
   }
}

proc new_title
DBMS BEGIN
   DBMS SQL INSERT INTO titles VALUES \
      (:+title_id, :+name, :+genre_code, \
      :+dir_last_name, :+dir_first_name, :+film_minutes, \
      :+rating_code, :+release_date, :+pricecat)
   DBMS SQL INSERT INTO title_dscr VALUES \
      (:+title_id, :+line_no, :+dscr_text)
   DBMS SQL INSERT INTO tapes VALUES \
      (:+title_id, :+copy_num, :+status, :+times_rented)
DBMS COMMIT
return 0
```

The procedure `tran_handle` is a generic handler for the application’s transactions. The procedure `new_title` contains the transaction statements. This method reduces the amount of error checking code.

The application executes the transaction by executing

```
call tran_handle ”new_title()”
```

The procedure `tran_handle` receives the argument “new_title” and writes it to the variable `subroutine`. It declares a JPL variable, `jpl_retcode`. After performing colon processing, `:subroutine` is replaced with its value, `new_title`, and JPL calls the procedure. The procedure `new_title` begins the transaction, performs three inserts, and commits the transaction.

If `new_title` executes without any errors, it returns 0 to the variable `jpl_retcode` in the calling procedure `tran_handle`. JPL then evaluates the `if` statement, displays a success message, and exits.

If however an error occurs while executing `new_title`, Prolifics calls the application's error handler. The error handler should display any error messages and return the abort code, 1.

For example, assume the first `INSERT` in `new_title` executes successfully but the second `INSERT` fails. In this case, Prolifics calls the error handler to display an error message. When the error handler returns the abort code 1, Prolifics aborts the procedure `new_title` (therefore, the third `INSERT` is not attempted). Prolifics returns 1 to `jpl_retcode` in the calling procedure `tran_handle`. JPL evaluates the `if` statement, displays a message, and executes a rollback. The rollback undoes the insert to the table `titles`.

Transaction Control on Multiple Cursors

SQL Server provides two-phase commit service for distributed transactions. In a two-phase commit, one main transaction controls two or more subtransactions on one or more servers. A subtransaction is a transaction on single cursor, like those described in the section above.

With two-phase commit service using Microsoft SQL Server, the commit server and the target server must be different.

The main transaction must be declared with this command:

```
DBMS [WITH CONNECTION connection] \
   DECLARE transaction-name TRANSACTION FOR \
   APPLICATION application SITES sites
```

- *connection*: if no connection is given, the default connection is used; the connection data structure stores a user login name, a server name, and an interface file name. Because SQL Server requires that a particular server be responsible for coordinating a two-phase commit, the connection declaration must include a server name.

- *transaction*: the name of the transaction; SQL Server does not permit periods (`.`) or colons (`;`) in a transaction name. Because `transaction` and `tran` are keywords for both Prolifics and SQL Server, do not use these words for this argument.

- *application*: the name of the application; it can be any character string that is not a keyword.
- *sites*: the number of cursors (i.e., subtransactions) participating in the two-phase commit. This value is used by the SQL Server commit and recovery systems and must be set appropriately.

After the transaction is declared, its name is used to begin and to commit or to rollback the transaction. The syntax is

`DBMS BEGIN` *transaction-name*

`DBMS COMMIT` *transaction-name*

`DBMS ROLLBACK` *transaction-name*

As with cursors and connections, Prolifics uses a data structure to manage a two-phase commit transaction. This structure should be closed when the transaction is completed. When the structure is closed, Prolifics calls the support routine to close the connection with the SQL Server commit service:

`DBMS CLOSE TRANSACTION` *transaction-name*

Operations on a single cursor are subtransactions. To control a subtransaction in a two-phase commit transaction, the following commands can be used:

`DBMS` *[* `WITH CURSOR` *cursor]* `BEGIN`

`DBMS` *[* `WITH CURSOR` *cursor]* `SAVE` *savepoint*

`DBMS` *[* `WITH CURSOR` *cursor]* `PREPARE_COMMIT`

`DBMS` *[* `WITH CURSOR` *cursor]* `COMMIT`

`DBMS` *[* `WITH CURSOR` *cursor]* `ROLLBACK` *[savepoint]*

The command `DBMS PREPARE_COMMIT` is an additional command required by the two-phase commit service. Executing it signals that the subtransaction has been performed and that the server is ready is to commit the update. After the application has “prepared” all the subtransactions, it issues a `COMMIT` to the main transaction and each subtransaction.

The sequence of events in a SQL Server two-phase commit transaction is the following:

- Declare any necessary connections and cursors.
- Declare the main transaction.

  ```
  DBMS DECLARE tname TRANSACTION FOR SITES sites \
     APPLICATION application
  ```

- Begin the main transaction.

  ```
  DBMS BEGIN tname
  ```

- For each subtransaction cursor, begin the subtransaction and execute the desired operations. When all subtransactions are complete, execute a `PREPARE_COMMIT` for each. In the pseudo code below there are three subtransactions (using `cursor1`, the default cursor, and `cursor2`):

  ```
  DBMS WITH CURSOR cursor1 BEGIN
  DBMS WITH CURSOR cursor1 EXECUTE USING parm

  DBMS BEGIN
  DBMS SQL statement
  DBMS SAVE savepoint
  DBMS SQL statement
  if error
     DBMS ROLLBACK savepoint
     DBMS SQL statement

  DBMS WITH CURSOR cursor2 BEGIN
  DBMS WITH CURSOR cursor2 EXECUTE USING parm

  DBMS WITH CURSOR cursor1 PREPARE_COMMIT
  DBMS PREPARE_COMMIT
  DBMS WITH CURSOR cursor2 PREPARE_COMMIT
  ```

- Commit the main transaction.

  ```
  DBMS COMMIT tname
  ```

- Commit each subtransaction indicating a named or default cursor.

  ```
  DBMS WITH CURSOR cursor1 COMMIT
  DBMS COMMIT
  DBMS WITH CURSOR cursor2 COMMIT
  ```

- Close the transaction.

  ```
  DBMS CLOSE TRANSACTION tname
  ```

It is strongly recommended that the application use an error handler while the transaction is executing. If an error occurs while executing a command in the subtransaction (i.e., executing a SQL statement or a named cursor), the application should not continue executing the transaction.

An example with an error handler follows.

```
########################################################
# Declare connections and specify servers.
DBMS DECLARE c1 CONNECTION \
   FOR USER :uid PASSWORD :pwd SERVER maple \
   INTERFACES ’/usr/sybase/interfaces.ny’
DBMS DECLARE c2 CONNECTION \
   FOR USER :uid PASSWORD :pwd SERVER juniper

# Declare cursors.
# Use :: to insert a value when the cursor is executed,
# not when the cursor is declared.
DBMS WITH CONNECTION c1 DECLARE x CURSOR FOR INSERT \
   emp (ss, last, first, street, city, st, zip, grade) \
   VALUES (::ss, ::last, ::first, ::street, ::city, \
   ::st, ::zip, ::grade)
DBMS WITH CONNECTION c2 DECLARE y CURSOR FOR INSERT \
   acc (ss, sal, exmp) VALUES (::ss, ::sal, ::exmp)

########################################################
proc 2phase
vars retval
retval = sm_s_val ()
if retval
{
   msg reset ”Invalid entry.”
   return
}
DBMS WITH CONNECTION c1 DECLARE new_emp TRANSACTION \
   FOR APPLICATION personnel SITES 2
DBMS ONERROR JPL tran_error
call do_tran
if !(retval)
   msg emsg ”Transaction succeeded.”
else
{
   DBMS ROLLBACK newemp
   if retval >= 100
      DBMS WITH CURSOR x ROLLBACK
   if retval >= 200
      DBMS WITH CURSOR y ROLLBACK
}
DBMS ONERROR CALL generic_errors
DBMS CLOSE TRANSACTION new_emp
return

proc do_tran
# Begin new_emp and set the flag tran_level (LDB variable)
DBMS BEGIN new_emp

   DBMS WITH CURSOR x BEGIN
   tran_level = ”1”
   DBMS WITH CURSOR x EXECUTE USING \
      ss, last, first, street, city, st, zip, grade

   DBMS WITH CURSOR y BEGIN
   tran_level = ”2”
   DBMS WITH CURSOR y EXECUTE USING \
      ss, startsal, exemptions

   DBMS WITH CURSOR x PREPARE_COMMIT
   DBMS WITH CURSOR y PREPARE_COMMIT

# Execute commits.
DBMS COMMIT new_emp
   DBMS WITH CURSOR x COMMIT
   DBMS WITH CURSOR y COMMIT

msg emsg ”Insert completed.”
tran_level = ””
return

########################################################
proc tran_error
vars fail_area [2](20), tran_err(3)
fail_area[1] = ”address”
fail_area[2] = ”accounting data”

if tran_level != ””
{
   # Display an error message describing the failure.
   msg emsg ”%WTransaction failed. Unable to insert \
         :fail_area[tran_level] because of ” @dmengerrmsg
   math tranerr = tran_level * 100
   tran_level = ””
   return :tranerr
}
msg emsg @dmengerrmsg
return 1
```

Transaction Manager Processing

Transaction Model for SQL Server

Each database driver contains a standard transaction model for use with the transaction manager. The transaction model is a C program which contains the main processing for each of the transaction manager commands. You can edit this program; however, be aware that the transaction model is subject to change with each release. For SQL Server, the name of the standard transaction model is `tmmss1.c`.

The standard transaction model for SQL Server calls `DBMS FLUSH` instead of `DBMS CANCEL` as part of the processing for the `FINISH` command. If a query has returned a very large select set, closing the screen might be longer with the `FLUSH` command. You can change this behavior by editing the model; however, the model is subject to change in future releases, so you should track your changes in order to update future versions.

Using Version Columns

For a SQL Server timestamp column, you can set the In Update Where and In Delete Where properties to Yes. This includes the value fetched to that widget in the SQL `UPDATE` and `DELETE` statements that are generated as part of the `SAVE` command.

SAVE Commands

If you specify a `SAVE` command with a table view parameter, it is called a partial command. A partial command is not applied to the entire transaction tree. In the standard transaction models, partial `SAVE` commands do not commit the database transaction. In order to save those changes, you must do an explicit `DBMS COMMIT`. Otherwise, those changes could be rolled back if the database engine performs an automatic rollback when the database connection is closed.

SQL Server-Specific Commands

Prolifics for SQL Server provides commands for SQL Server-specific features. This section contains a reference page for each command. If you are using multiple engines or are porting an application to or from another engine, please note that these commands may work differently or may not be supported on some engines.

Using Browse Mode

`BROWSE`	Execute a `SELECT` for browsing.
`UPDATE`	Update a table while browsing.

Using Scrolling

`BUFFER_DEFAULT`	Set buffer size for scrolling for entire application.
`SET_BUFFER`	Control availability of SQL Server-based scrolling for `DBMS CONTINUE_BOTTOM`, `DBMS CONTINUE_TOP`, `DBMS CONTINUE_UP`.

Using Stored Procedures

`CANCEL`	Abort execution of a stored procedure.
`DECLARE CURSOR FOR RPC`	Declare a cursor to execute a stored procedure using a remote procedure call.
`FLUSH`	Abort execution of a stored procedure.
`NEXT`	Execute the next statement in a stored procedure.
`SET`	Set execution behavior for a procedure (execute all, stop at fetch, etc.).
`TYPE`	Set data types for parameters of a stored procedure executed with an rpc cursor.

Using Transactions

`BEGIN`	Begin a transaction.
`CLOSE_ALL_TRANSACTIONS`	Close all transactions declared for two-phase commit.
`CLOSE_TRANSACTION`	Close a two–phase transaction.
`COMMIT`	Commit a transaction.
`DECLARE TRANSACTION`	Declare a transaction for two-phase commit.
`PREPARE_COMMIT`	Indicate that a subtransaction is ready to commit.
`ROLLBACK`	Rollback a transaction.
`SAVE`	Set a savepoint in a transaction.

BEGIN

Start a transaction

```
DBMS [WITH CONNECTION connection-name] BEGIN
   DBMS [WITH CURSOR cursor-name] BEGIN

DBMS [WITH CONNECTION connection-name] BEGIN two-phase-transaction-name
```

two-phase-transaction-name	Specify an existing two phase transaction.
`WITH CURSOR` *cursor-name*	Specify the named cursor for the transaction. If no `WITH CURSOR` or `WITH CONNECTION` clause is used, Prolifics begins a transaction on the default cursor of the default connection.
`WITH CONNECTION` *connection-name*	If a `WITH CONNECTION` clause is used, Prolifics begins a transaction on the default cursor of the named connection. If no `WITH CURSOR` or `WITH CONNECTION` clause is used, Prolifics begins a transaction on the default cursor of the default connection.

Description

This command sets the starting point of a transaction. It is available in two contexts. It can start a transaction on a single cursor, or it can start a distributed transaction that can involve multiple cursors on one or more servers.

A transaction is a logical unit of work on a database contained within `DBMS BEGIN` and `DBMS COMMIT` statements. `DBMS BEGIN` defines the start of a transaction. After a transaction is begun, changes to the database are not committed until a `DBMS COMMIT` is executed. Changes are undone by executing `DBMS ROLLBACK`.

To begin a distributed transaction (two-phase transaction), first declare a named transaction with `DBMS DECLARE TRANSACTION`. Because this statement supports a `WITH CONNECTION` clause, Prolifics associates the transaction name with a particular connection; the connection's server is the coordinating server for the distributed transaction. When the application executes `DBMS BEGIN` *transaction-name* where *transaction-name* is the name of the declared transaction, Prolifics starts the transaction on the coordinating server.

Be sure to terminate the transaction with a `DBMS ROLLBACK` or `DBMS COMMIT` before logging off. Note that Prolifics will not close a connection with a pending two-phase commit transaction.

Example

Refer to the example in Using Transactions on page 25.

See Also

Using Transactions on page 25

`CLOSE_ALL_TRANSACTIONS`

`CLOSE TRANSACTION`

`COMMIT`

`DECLARE TRANSACTION`

`PREPARE_COMMIT`

`ROLLBACK`

`SAVE`

BROWSE

Retrieve SELECT results one row at a time

```
DBMS BROWSE SELECTstmt
```

Description

This command allows an application to execute a `SELECT` in “browse” mode. This means that SQL Server will return the `SELECT` rows one at a time to the Prolifics application; SQL Server will not set any shared locks for the `SELECT`. The application can use the companion command `DBMS UPDATE` to update the current row. SQL Server will verify that the row has not been changed before it issues the `UPDATE`.

To update in browse mode, the table being updated must have a timestamp column and a unique index. A row’s timestamp indicates the last time the row was updated. If the timestamp has not changed since `DBMS BROWSE` was executed, the application can update the row. If the timestamp has changed, then some other user or application has updated the row after `DBMS BROWSE` was executed. The update is aborted and an error is returned.

Browse mode requires a connection with two default cursors. The application must open the browse mode connection by setting the `CURSORS` option to 2. Prolifics uses one default cursor to select the rows and the other default cursor to update the rows.

It is the programmer’s responsibility to determine whether a table is browsable. If the table is not browsable, Prolifics returns the `DM_BAD_ARGS` error. If a table is browsable, Prolifics returns the first row in the select set when `DBMS BROWSE` is executed. Note that only one row is returned at a time.

To view the next row, the application must execute `DBMS CONTINUE`.

See Also

`CONTINUE`

`FLUSH`

`UPDATE`

BUFFER_DEFAULT

Specifies setting for engine-based non-sequential scrolling

```
DBMS [ WITH CONNECTION connection-name ] BUFFER_DEFAULT value
```

value	Specifies the size of the buffer for SQL Server-based scrolling, if it is non-zero.

Description

A Prolifics application can use either Prolifics-based or SQL Server-based scrolling to execute `DBMS CONTINUE`, `DBMS CONTINUE_TOP`, `DBMS CONTINUE_UP`, and `DBMS CONTINUE_BOTTOM`.

The size of the buffer is determined by the value specified with these commands.

See Also

`SET_BUFFER`

CANCEL

Cancel the execution of a stored procedure or discard select rows

`DBMS` *[* `WITH CURSOR` *cursor-name]* `CANCEL`

`WITH CURSOR` *cursor-name*	Specify a named cursor for the command. If this clause is not included, Prolifics issues the command on the default cursor of the default connection.

Description

This command cancels any outstanding work on the named cursor. In particular, this command can be used to cancel a pending stored procedure or discard unwanted select rows. When the statement is executed, the following operations are performed:

- Any rows to be fetched are flushed.
- Any remaining unexecuted statements are ignored.
- The procedure's return status code is returned.

Prolifics calls the SQL Server routine `dbcancel()` to perform this operation.

If the `WITH CURSOR` clause is not used, Prolifics executes the command on the default cursor.

See Also

Using Stored Procedures on page 17

`FLUSH`

CLOSE_ALL_TRANSACTIONS

Close all transactions declared for two-phase commit

```
DBMS CLOSE_ALL_TRANSACTIONS
```

Description

This command attempts to close all transactions declared for two-phase commit with `DBMS DECLARE TRANSACTION`. If the transaction has not been terminated by a `COMMIT` or `ROLLBACK`, Prolifics will return the error `DM_TRAN_PENDING`.

Prolifics will not close a connection unless all two-phase commit transactions have been closed. Furthermore, Prolifics will not close a two-phase commit transaction unless the application explicitly terminated the transaction with a `DBMS COMMIT` *transaction-name* or `DBMS ROLLBACK` *transaction-name*.

This helps prevent the application from terminating with a pending two-phase transaction. For if this happens, SQL Server marks the transaction's process as "infected." You will need the system administrator to delete the infected process.

Because this command verifies that all two-phase commit transactions were terminated, you must call this command before logging off.

Example

The JPL procedure `cleanup` checks that all declared transactions have been closed before closing the database connections. If there is a transaction pending, the error handler calls the JPL procedure `cleanup_failure`, which in turn calls the procedure `tran_cleanup`.

```
proc cleanup
   DBMS ONERROR JPL cleanup_failure
   DBMS CLOSE_ALL_TRANSACTIONS
   DBMS CLOSE_ALL_CONNECTIONS
   return

# Control string for APP1 is:
# APP1 = ^tran_cleanup

proc cleanup_failure (stmt, engine, flag)
   if @dmretcode == DM_TRAN_PENDING
   {
      call jm_keys APP1
   }
   return 0
```

```
proc tran_cleanup
   DBMS WITH CURSOR c1 ROLLBACK
   DBMS WITH CURSOR c2 ROLLBACK
   DBMS ROLLBACK tr1
   DBMS CLOSE TRANSACTION tr1
   return
```

See Also

Using Transactions on page 25

`BEGIN`

`CLOSE TRANSACTION`

`COMMIT`

`DECLARE TRANSACTION`

`ROLLBACK`

CLOSE TRANSACTION

Close a declared transaction structure

`DBMS CLOSE TRANSACTION` *two-phase-transaction-name*

two-phase-transaction -name	Specify an existing two phase transaction.

Description

This command closes the main transaction that was previously defined using `DBMS DECLARE TRANSACTION`. A main transaction controls the execution of a two-phase commit process. This command signals the completion of the main transaction and closes the SQL Server structures associated with the transaction.

An error code is returned if a transaction was pending. An application cannot close a connection with an open transaction.

See Also

Using Transactions on page 25

`BEGIN`

`CLOSE_ALL_TRANSACTIONS`

`COMMIT`

`DECLARE TRANSACTION`

`PREPARE_COMMIT`

`ROLLBACK`

`SAVE`

COMMIT

Commit a transaction

```
DBMS [ WITH CONNECTION connection-name ] COMMIT

DBMS [ WITH CURSOR cursor-name ] COMMIT


DBMS COMMIT two_phase_transaction_name
```

`WITH CONNECTION` *connection-name*	Specify the connection for this command. If the command does not contain a `WITH CONNECTION` clause, Prolifics issues the commit on the default connection.
`WITH CURSOR` *cursor-name*	Specify a named cursor for the command.
two-phase-transaction -name	Specify an existing two phase transaction.

Description

Use this command to commit a pending transaction. Committing a transaction saves all the work since the last `COMMIT`. Changes made by the transaction become visible to other users. If the transaction is terminated by `ROLLBACK`, the updates are not committed, and the database is restored to its state prior to the start of the transaction.

This command is available in two contexts. It can commit a transaction on a single cursor or it can commit a two-phase commit transaction. If a `WITH CURSOR` clause is used in a `DBMS COMMIT` statement, Prolifics commits the transaction on the named cursor. If a `WITH CONNECTION` clause is used, Prolifics commits the transaction on the default cursor of the named connection. If no `WITH` clause or no distributed transaction name is used, Prolifics commits the transaction on the default cursor of the default connection.

This command is available depending on the setting of various parameters in your environment. Refer to the section on transactions and your documentation for more information.

If a distributed transaction name is used, Prolifics issues the commit to the coordinating server. If this is successful, the application should issue a `DBMS COMMIT` for each subtransactions. A `WITH CURSOR` or `WITH CONNECTION` clause is required for a subtransaction on a named cursor or a subtransaction on the default cursor of a non-default connection. A `WITH CONNECTION` clause is required for a subtransaction on a named connection.

Example

Refer to the example in Using Transactions on page 25.

See Also

Using Transactions on page 25

`BEGIN`

`CLOSE TRANSACTION`

`DECLARE TRANSACTION`

`PREPARE_COMMIT`

`ROLLBACK`

`SAVE`

DECLARE CURSOR FOR RPC

Declare a named cursor for a remote procedure

```
DBMS [ WITH CONNECTION connection-name ] DECLARE cursor-name CURSOR FOR RPC \
    procedure [ ::parameter [ OUT ] [ data-type ] [ , ::parameter [ OUT ] [ data-type ] ... ] ]
```

`WITH CONNECTION` *connection-name*	Specify the connection for this command. If this clause is not included, Prolifics associates the cursor with the default connection.

Description

Use this command to create or redeclare a named cursor to execute a remote procedure call (rpc). Because Prolifics uses its binding mechanism to support rpc's, the default cursor cannot execute an rpc.

The keyword `RPC` is required. Following the keyword is the name of the procedure and the names of the procedure's parameters. All parameters must begin with a double colon, which is the Prolifics syntax for cursor parameters. If a parameter is an output parameter, the keyword `OUT` should follow the parameter name if the application is to receive its value. A parameter's data type can be given in the `DBMS DECLARE CURSOR` statement, or in a `DBMS TYPE` statement. Parameter names in the `DECLARE CURSOR` statement must exactly match the parameter names defined by the stored procedure.

The application executes an rpc cursor as it executes any named cursor, with `DBMS EXECUTE`.

Example

Refer to the example in Using Stored Procedures on page 17.

See Also

Using Stored Procedures on page 17

`@dmengreturn`

`CLOSE CURSOR`

`EXECUTE`

`TYPE`

DECLARE TRANSACTION

Declare a named transaction

```
DBMS [WITH CONNECTION connection-name] DECLARE transaction-name TRANSACTION FOR \
   SITES sites APPLICATION application
```

application	Optional argument that identifies the name of the transaction.
sites	Determines the number of subtransactions involved in the distributed transaction. Each cursor where a `BEGIN` is issued is a subtransaction. This number is critical to recovery if the transaction fails.
transaction-name	Specify the name of the two-phase commit transaction. Do not use the keywords “tran” or “transaction” for this argument. The application must use this name to begin, to commit or rollback, and to close the transaction.
`WITH CONNECTION` *connection-name*	Identify the server that will coordinate the distributed transaction. If the clause is not used, the server of the default connection is used. Be sure to name the server when declaring the connection.

Description

This command declares a two-phase commit transaction structure. After declaring the transaction, start the transaction using `DBMS BEGIN` and include the transaction name in that command. When the transaction is complete, close the transaction using either `DBMS CLOSE TRANSACTION` or `DBMS CLOSE_ALL_TRANSACTIONS`. An application must close all declared transactions before closing their connections.

Example

Refer to the example in Using Transactions on page 25.

See Also

`BEGIN`

`CLOSE_ALL_TRANSACTIONS`

`CLOSE TRANSACTION`

FLUSH

Flush any selected rows not fetched to Prolifics variables

`DBMS` *[* `WITH CURSOR` *cursor-name]* `FLUSH`

`WITH CURSOR` *cursor-name*	Specify a named cursor for the command. If this clause is not included, Prolifics issues the command on the default cursor of the default connection.

Description

Use this command to throw away any unread rows in the select set of the default or named cursor.

This command is often useful in applications that execute a stored procedure. If the stored procedure executes a `SELECT`, the procedure will not return the `DM_END_OF_PROC` signal if the select set is pending. The application can execute `DBMS CONTINUE` until the `DM_NO_MORE_ROWS` signal is returned, or it can execute `DBMS FLUSH`, which cancels the pending rows.

This command is also useful with queries that fetch very large select sets. The application can execute `DBMS FLUSH` after executing the `SELECT`, or after a defined time-out interval. This guarantees a release of the shared locks on all the tables involved in the fetch. Of course, after the rows have been flushed, the application cannot use `DBMS CONTINUE` to view the unread rows.

Prolifics calls the SQL Server routine `dbcanquery()` to perform this operation.

Example

```
proc large_select
# Do not allow the user to see any more rows than
# can be held by the onscreen arrays.
DBMS SQL SELECT * FROM titles
if @dmretcode != DM_NO_MORE_ROWS
   DBMS FLUSH
return 0
```

See Also

`DECLARE CURSOR`

`CANCEL`

`CONTINUE`

`NEXT`

NEXT

Execute the next statement in a stored procedure

`DBMS` *[* `WITH CURSOR` *cursor-name]* `NEXT`

`WITH CURSOR` *cursor-name*	Specify a named cursor for the command. If this clause is not included, Prolifics issues the command on the default cursor of the default connection.

Description

Unless `DBMS SET` equals `EXECUTE_ALL`, an application must execute `DBMS NEXT` after a stored procedure returns one or more `SELECT` rows to Prolifics. `DBMS NEXT` executes the next statement in the stored procedure. If the application executes `DBMS NEXT` and there are no more statements to execute, Prolifics returns the `DM_END_OF_PROC` code.

If a cursor is associated with two or more SQL statements and `DBMS SET` equals `STOP_AT_FETCH`, the application must execute `DBMS NEXT` after each `SELECT` that returns rows to Prolifics. If `DBMS SET` equals `SINGLE_STEP`, the application must execute `DBMS NEXT` after each statement, including non-`SELECT` statements. If the application executes `DBMS NEXT` after all of the cursor's statements have been executed, Prolifics returns the `DM_END_OF_PROC` code.

Example

Refer to the example in Using Stored Procedures on page 17.

See Also

Using Stored Procedures on page 17

`DECLARE CURSOR`

`CANCEL`

`CONTINUE`

`FLUSH`

`SET` *[* `EXECUTE_ALL | SINGLE_STEP | STOP_AT_FETCH` *]*

PREPARE_COMMIT

Prepare a two phase commit

`DBMS` *[* `WITH CURSOR` *cursor-name]* `PREPARE_COMMIT`

`WITH CURSOR` *cursor-name*	Specify a named cursor for the command. If this clause is not included, Prolifics issues the command on the default cursor of the default connection.

Description

Use of this command is required during the two-phase commit service. It is executed for each subtransaction when the subtransaction has been performed. Execution of this command signals the application that the server is ready to commit the update. After the application has “prepared” all the subtransactions, it needs to issue a `DBMS COMMIT` to the main transaction and to each subtransaction.

If the `WITH CURSOR` clause is not used, Prolifics issues the command on the default cursor.

Example

Refer to the example in Using Transactions on page 25.

See Also

Using Transactions on page 25

`BEGIN`

`CLOSE TRANSACTION`

`COMMIT`

`DECLARE TRANSACTION`

`ROLLBACK`

`SAVE`

ROLLBACK

Roll back a transaction

`DBMS` *[* `WITH CONNECTION` *connection-name]* `ROLLBACK` *savepoint*

`DBMS` *[* `WITH CURSOR` *cursor-name]* `ROLLBACK` *savepoint*

`DBMS ROLLBACK` *two_phase_transaction_name*

`WITH CONNECTION` *connection-name*	Specify the connection for this command. If the command does not contain a `WITH CONNECTION` clause, Prolifics issues the rollback on the default connection.
`WITH CURSOR` *cursor-name*	Specify a named cursor for the command. If this clause is not included, Prolifics issues the command on the default cursor of the default connection.
savepoint	If included, only the statements that were issued after the specified savepoint are rolled back.
two-phase-transaction -name	Specify an existing two phase transaction.

Description

Use this command to rollback a transaction and restore the database to its state prior to the start of the transaction or at the time of the specified savepoint.

This command is available in two contexts. It can rollback a transaction on a single cursor, or it can rollback a two-phase rollback transaction. If a `WITH CURSOR` clause is used in a `DBMS ROLLBACK` statement, Prolifics rolls back the transaction on the named cursor. If a `WITH CONNECTION` clause is used, Prolifics rolls back the transaction on the default cursor of the named connection. If no `WITH` clause or no distributed transaction name is used, Prolifics rolls back the transaction on the default cursor of the default connection.

Example

Refer to the example in Using Transactions on page 25.

If a distributed transaction name is used, Prolifics issues the rollback to the coordinating server. The application should also issue a `DBMS ROLLBACK` for each subtransaction. A `WITH CURSOR` or `WITH CONNECTION` clause is required for a subtransaction on a named cursor or a subtransaction on the default cursor of a non-default connection.

See Also

Using Transactions on page 25

`BEGIN`

`COMMIT`

`DECLARE TRANSACTION`

`PREPARE_COMMIT`

`SAVE`

SAVE

Set a savepoint within a transaction

```
DBMS [WITH CONNECTION connection-name] SAVE savepoint
   DBMS [WITH CURSOR cursor-name] SAVE savepoint
```

savepoint	Specify the name of the savepoint.
`WITH CURSOR` *cursor-name*	Specify a named cursor for the command. If this clause is not included, Prolifics issues the command on the default cursor of the default connection.

Description

This command creates a savepoint in the transaction. A savepoint is a place-marker set by the application within a transaction. When a savepoint is set, the statements following the savepoint can be cancelled using `DBMS ROLLBACK` *savepoint*. A transaction can have multiple savepoints.

When the transaction is rolled back to a savepoint, the transaction must then either be completed *or* rolled back to the beginning.

This feature is useful for any long, complicated transaction. For example, an order entry application might involve many screens where an end-user must enter data regarding the order. As the user completes each screen, the application can issue a savepoint. Therefore, if an error occurs on the fifth screen, the application can simply rollback the procedures on the fifth screen.

Example

```
proc new_title
   DBMS SQL INSERT INTO titles VALUES \
      (:+title_id, :+name, :+genre_code, \
      :+dir_last_name, :+dir_first_name, :+film_minutes, \
      :+rating_code, :+release_date, :+pricecat)
   DBMS SAVE s1
call new_dscr
call new_tapes
DBMS COMMIT
return 0

proc new_dscr
   DBMS SQL INSERT INTO title_dscr VALUES \
      (:+title_id, :+line_no, :+dscr_text)
   DBMS SAVE s2
return 0
```

```
proc new_tapes
   DBMS SQL INSERT INTO tapes VALUES \
      (:+title_id, :+copy_num, :+status, :+times_rented)
return 0
```

See Also

Using Transactions on page 25

`BEGIN`

`COMMIT`

`DECLARE TRANSACTION`

`PREPARE_COMMIT`

`ROLLBACK`

SET

Set handling for a cursor that executes a stored procedure or multiple statements

```
DBMS [WITH CURSOR cursor-name] SET EXECUTE_ALL

DBMS [WITH CURSOR cursor-name] SET SINGLE_STEP

DBMS [WITH CURSOR cursor-name] SET STOP_AT_FETCH
```

`WITH CURSOR` *cursor-name*	Specify a named cursor for the command. If this clause is not included, Prolifics issues the command on the default cursor of the default connection.

Description

This command controls the execution of a stored procedure or a cursor that contains multiple SQL statements. This command allows the following options:

`EXECUTE_ALL`

Specifies that the DBMS return control to Prolifics only when all statements have been executed or when an error occurs. If a SQL `SELECT` is executed, only the first pageful of rows is returned to Prolifics variables. This option can be set for a multi-statement or a stored procedure cursor.

`SINGLE_STEP`

Specifies that the DBMS return control to Prolifics after executing each statement belonging to the multi-statement cursor. After each `SELECT`, the user can press a function key to execute a `DBMS CONTINUE` and scroll the select set. To resume executing the cursor's statements, the application must execute `DBMS NEXT`. This option can be set for a multi-statement cursor. If this option is used with a stored procedure cursor, Prolifics uses the default setting `STOP_AT_FETCH`.

`STOP_AT_FETCH`

Specifies that the DBMS return control to Prolifics after executing a SQL `SELECT` that fetches rows. (Note that control is not returned for a `SELECT` that assigns a value to a local SQL Server parameter.) The application can use `DBMS CONTINUE` to scroll through the select set. To resume executing the cursor's statements or procedure, the application must execute `DBMS NEXT`. This option can be set for a multi-statement or a stored procedure cursor.

The default behavior for both stored procedure and multi-statement cursors is STOP_AT_FETCH. Executing DBMS SET with no arguments restores the default behavior.

Example

```
DBMS DECLARE x CURSOR FOR \
SELECT cust_id, first_name, last_name, member_status \
   FROM customers WHERE cust_id = ::cust_id \
INSERT INTO rentals (cust_id, title_id, copy_num, \
   rental_date, price) \
   VALUES (::cust_id, ::title_id, ::copy_num, \
   ::rental_date, ::price)

msg d_msg ”%KPF1 START %KPF2 SCROLL SELECT\
 %KPF3 EXECUTE NEXT STEP”

proc f1
# This function is called by the PF1 key.
DBMS WITH CURSOR x SET_BUFFER 10
DBMS WITH CURSOR x SET SINGLE_STEP
DBMS WITH CURSOR x EXECUTE USING cust_id, cust_id, \
   title_id, copy_num, rental_date, price
DBMS WITH CURSOR x SET
return

proc f2
# This function is called by the PF2 key.
DBMS WITH CURSOR x CONTINUE
if @dmretcode == DM_NO_MORE_ROWS
   msg emsg ”All rows displayed.”
return

proc f3
# This function is called by the PF3 key.
DBMS WITH CURSOR x NEXT
if @dmretcode == DM_END_OF_PROC
   msg emsg ”Done!”
return
```

See Also

Using Stored Procedures on page 17

CANCEL

CONTINUE

DECLARE CURSOR

DECLARE CURSOR FOR EXEC

DECLARE CURSOR FOR RPC

```
FLUSH

NEXT
```

SET_BUFFER

Use engine-based scrolling

`DBMS` *[* `WITH CURSOR` *cursor-name]* `SET_BUFFER` *[number-of-rows]*

`WITH CURSOR` *cursor-name*	Specify a named cursor for the command. If this clause is not included, Prolifics issues the command on the default cursor of the default connection.

Description

There are two methods of using the non-sequential scrolling commands `DBMS CONTINUE_BOTTOM`, `DBMS CONTINUE_TOP`, and `DBMS CONTINUE_UP`. In one method, an application uses Prolifics-based scrolling by setting up a continuation file with `DBMS STORE FILE`. In the other method, an application uses SQL Server-based scrolling by setting a flag for a cursor with `DBMS SET_BUFFER`.

SQL Server supports non-sequential scrolling if the application has set up a buffer for result rows. This command sets the `SELECT` cursor to use SQL Server-based scrolling. If an application does not need `DBMS CONTINUE_UP` or is using a continuation file (`DBMS STORE FILE`), this command is *not* needed.

If the `WITH CURSOR` clause is used, Prolifics sets the flag for the named cursor. If the `WITH CURSOR` clause is not used, Prolifics sets the flag for the default `SELECT` cursor.

number-of-rows is the number of rows SQL Server will buffer. To be useful, *number-of-rows* should be greater than the number of occurrences in the Prolifics destination fields.

When this command is used with a `SELECT` cursor, SQL Server saves the specified number of result rows in memory. When the application executes `DBMS CONTINUE_BOTTOM`, `DBMS CONTINUE_TOP`, or `DBMS CONTINUE_UP` commands, the result rows in memory are returned.

The buffer is maintained for the life of the cursor, or until the buffer is released with this command:

`DBMS` *[* `WITH CURSOR` *cursor-name*`] SET_BUFFER`

Executing the command without supplying the *number-of-rows* argument turns off the feature for the named or default cursor and frees the buffer. Note that re-declaring the cursor does not free the buffer. Closing the cursor does release the buffer.

Because the use of this command is expensive (approximately 2K of memory per row), it should be used only if the application needs non-sequential scrolling but cannot use scrolling arrays or a continuation file. The application should turn off `DBMS SET_BUFFER` when finished with the select set.

Note the following restrictions:

- Only a few engines support native scrolling. Therefore, this command might not be supported with other engines. Prolifics-based scrolling is supported on all engines with `DBMS STORE FILE`.
- Each `DBMS CONTINUE_BOTTOM`, `DBMS CONTINUE_TOP`, and `DBMS CONTINUE_UP` requires a trip to the server. With Prolifics-based scrolling, the rows are fetched once. When the application attempts to view rows already fetched, Prolifics reads them from the continuation file rather than requesting them from the server.

Example

```
DBMS DECLARE t_cursor CURSOR FOR SELECT * FROM titles
DBMS WITH CURSOR t_cursor SET_BUFFER 500

proc scroll_up
DBMS WITH CURSOR t_cursor CONTINUE_UP
return

proc scroll_down
DBMS WITH CURSOR t_cursor CONTINUE_DOWN
return
```

See Also

`CONTINUE_BOTTOM`

`CONTINUE_TOP`

`CONTINUE_UP`

`STORE`

TRANSACTION

Set a default transaction for use in two-phase commits

```
DBMS TRANSACTION two-phase-transaction-name
```

Description

If an application has declared more than one two-phase commit transaction, it can use this command to set the default two-phase commit transaction for a subtransaction.

See Also

`BEGIN`

`COMMIT`

`DECLARE TRANSACTION`

`PREPARE_COMMIT`

`ROLLBACK`

`SAVE`

TYPE

Declare parameter data types for an rpc cursor

`DBMS WITH CURSOR` *cursor-name* `TYPE` *parameter data-type [, parameter data-type ...]*

`WITH CURSOR` *cursor-name*	Specify a named cursor for the command.

Description

If an application has declared a cursor for a remote procedure call ("rpc") but has not declared the data types of the procedure's parameters, it should use the `DBMS TYPE` command.

parameter is the name of a parameter in the stored procedure and in the `DBMS DECLARE CURSOR` statement. *data-type* is the data type of the parameter in the stored procedure. Prolifics uses the information supplied with this command to execute the remote procedure call. Please note that these data types have no effect on any data formatting performed by colon-plus processing or binding.

Executing this command with no arguments deletes all type information for the named cursor.

Example

```
########################################################
#procedure newprice:
#create proc newprice @pricecat char(1), @percent float,
# @price money output, @proposed_price money output
# as
#  select @price = (select price from pricecats
#    where pricecat = @pricecat)
#  select @proposed_price = @price + (@price * @percent)
########################################################

DBMS DECLARE nc CURSOR FOR \
   RPC newprice ::pricecat, ::percent, ::price OUT, \
    ::proposed_price OUT

DBMS WITH CURSOR nc TYPE \
   percent float, price money, proposed_price money

DBMS WITH CURSOR nc EXECUTE \
   USING pricecat, percent, price, proposed_price
```

See Also

Using Stored Procedures on page 17

`DECLARE CURSOR FOR RPC`

UPDATE

Update a table while browsing

```
DBMS UPDATE table-name SET column = value [ , column = value ... ]
```

Description

Browse mode permits an application to browse through a select set, updating a row at a time. Browse mode is useful for an application that wants to ensure that a row has not been changed in the interval between the fetch and the update of the row.

When `DBMS BROWSE` is executed, it fetches the rows in the select set one at a time. The application should provide other JPL procedures to execute `DBMS CONTINUE` and `DBMS UPDATE` commands.

Please note that the `DBMS UPDATE` statement has no `WHERE` clause. Prolifics calls a SQL Server routine to build a `WHERE` clause using the unique index of the current row and the value of its timestamp column when the row was fetched. If the timestamp value has not been changed, the row is updated. However, if the timestamp value has changed, then another user has modified the row since the application executed `DBMS BROWSE`. In this case, SQL Server will not perform the update.

Example

Refer to the manual page for `BROWSE`.

See Also

`BROWSE`

`CANCEL`

`CONTINUE`

`FLUSH`

USE

Open an existing database

`DBMS` *[* `WITH CONNECTION` *connection-name]* `USE` *database-name*

`WITH CONNECTION` *connection-name*	Specify the connection for this command. If this clause is not included, Prolifics issues the command on the default connection.
database-name	Specify an existing database.

Description

This command changes a connection's default database. *database-name* must reference an existing database, and the user must have the appropriate permissions to access the database or else Prolifics returns an error.

Example

```
DBMS DECLARE c1 CONNECTION FOR \
   USER ':uname' PASSWORD ':pword' SERVER ':server' \
   DATABASE 'videobiz'
DBMS SQL SELECT * FROM titles
DBMS WITH CONNECTION c1 USE projects
DBMS SQL SELECT * FROM newjobs
```

See Also

Connecting to a Database Engine on page 5

Command Directory for SQL Server

The following table lists all commands available in Prolifics's database driver for SQL Server. Commands available to all database drivers are described in the *Programming Guide*.

Table 2. Commands for SQL Server

Command Name	Description	Documentation Location
`ALIAS`	Name a Prolifics variable as the destination of a selected column or aggregate function	*Programming Guide*
`BEGIN`	Begin a transaction	page 35
`BINARY`	Create a Prolifics variable for fetching binary values	page 810
`BROWSE`	Execute a SQL `SELECT` for browsing	page 37
`BUFFER_DEFAULT`	Set the size of the buffer for engine-based scrolling	page 38
`CANCEL`	Abort execution of a stored procedure	page 39
`CATQUERY`	Redirect select results to a file or a Prolifics variable	
`CLOSE_ALL_CONNECTIONS`	Close all connections on all engines	
`CLOSE_ALL_TRANSAC-TIONS`	Close all transactions	page 40
`CLOSE CONNECTION`	Close a named connection	
`CLOSE CURSOR`	Close a named cursor	
`CLOSE TRANSACTION`	Close a named transaction	page 42
`COLUMN_NAMES`	Return the column name, not column data, to a Prolifics variable	
`COMMIT`	Commit a transaction	page 43

Command Name	Description	Documentation Location
`CONNECTION`	Set a default connection and engine for the application	
`CONTINUE`	Fetch the next screenful of rows from a select set	*Database Guide* & *Database Drivers*
`CONTINUE_BOTTOM`	Fetch the last screenful of rows from a select set	*Database Guide* & *Database Drivers*
`CONTINUE_DOWN`	Fetch the next screenful of rows from a select set	*Database Guide* & *Database Drivers*
`CONTINUE_TOP`	Fetch the first screenful of rows from a select set	*Database Guide* & *Database Drivers*
`CONTINUE_UP`	Fetch the previous screenful of rows from a select set	*Database Guide* & *Database Drivers*
`DECLARE CONNECTION`	Declare a named connection to an engine	*Database Guide* & *Database Drivers*
`DECLARE CURSOR`	Declare a named cursor	*Database Guide* & *Database Drivers*
`DECLARE CURSOR FOR RPC`	Declare a cursor to execute a stored procedure using a remote procedure call	page 45
`DECLARE TRANSACTION`	Declare a transaction for two phase commit	page 46
`ENGINE`	Set the default engine for the application	
`EXECUTE`	Execute a named cursor	
`FLUSH`	Flush any selected rows	page 47
`FORMAT`	Format the results of a `CAT-QUERY`	
`NEXT`	Execute the next statement in a stored procedure	page 49
`OCCUR`	Set the number of rows for Prolifics to fetch to an array and set the occurrence where Prolifics should begin writing result rows	

Command Name	Description	Documentation Location
`ONENTRY`	Install a JPL procedure or C function that Prolifics will call before executing a `DBMS` statement	
`ONERROR`	Install a JPL procedure or C function that Prolifics will call when a `DBMS` statement fails	*Database Guide* & *Database Drivers*
`ONEXIT`	Install a JPL procedure or C function that Prolifics will call after executing a `DBMS` statement	
`PREPARE_COMMIT`	Indicate that a transaction is ready to commit	page 50
`ROLLBACK`	Roll back a transaction	page 51
`SAVE`	Set a savepoint in a transaction	page 53
`SET` *parameter*	Set execution behavior for a stored procedure	page 55
`SET_BUFFER`	Set engine-based scrolling for a cursor	page 58
`START`	Set the first row for Prolifics to return from a select set	
`STORE`	Store the rows of a select set in a temporary file so the application can scroll through the rows	
`TRANSACTION`	Set the default transaction	page 60
`TYPE`	Set data types for parameters of a stored procedure executed with an rpc cursor	page 61
`UNIQUE`	Suppress repeating values in a selected column	
`UPDATE`	Update a table while browsing	page 62
`USE`	Open an existing database	page 63

Command Name	Description	Documentation Location
`WITH CONNECTION`	Specify the connection to use for a command	
`WITH CURSOR`	Specify the cursor to use for a command	
`WITH ENGINE`	Specify the engine to use for a command	