JAM/DB*i* for SYBASE

August 19, 1992

Notes for SYBASE

This appendix provides documentation specific to SYBASE.

It discusses the following:

- engine initialization
- connection declaration
- cursors
- formatting for colon-plus and binding
- errors and warnings
- utilities
- engine-specific features
- command directory for JAM/DB*i* SYBASE

This document is designed as a supplement, not a replacement, to the JAM/DB*i* manual. Each section identifies its companion chapter or section in the JAM/DB*i* manual.

1.1 ENGINE INITIALIZATION *See* JAM/DB*i* *Manual – Section 7.1*

By default, JAM/DB*i* uses the following values in `dbiinit.c` for SYBASE initialization:

```
static vendor_t vendor_list[] =
{
    {"sybase", dm_sybsup, DM_PRESERVE_CASE ,(char *) 0},

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

The default settings are as follows:

`sybase`	Engine name. May be changed.
`dm_sybsup`	Support routine name. Do not change.
`DM_PRESERVE_CASE`	Case setting for matching SELECT columns with JAM variable names. May be changed.

1.1.1 Engine Name and Support Routine

An application may change the engine name associated with the support routine `dm_sybsup`. 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, make the following change:

```
static vendor_t vendor_list[] =
{
    {"tracking", dm_sybsup, DM_PRESERVE_CASE, (char *) 0},

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

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

`dbms ENGINE` ***sybase_engine_name***

where ***sybase_engine_name*** is the string used in `vendor_list`. For example,

```
dbms ENGINE sybase
```

or

```
dbms ENGINE tracking
```

`dm_sybsup` is the name of the support routine for SYBASE. This name should not be changed.

If your application is using multiple engines, you need to add a line to `vendor_list` for each engine. You also need to modify your makefile to support both engines and recompile the JAM/DB*i* executables, `jxdbi` and `jamdbi`.

1.1.2 Case and Error Flags

The case flag, `DM_PRESERVE_CASE`, determines how JAM/DB*i* uses case when searching for JAM variables for holding SELECT results. JAM/DB*i* uses this setting when comparing SYBASE column names to either a JAM variable name or to a column name in a DBMS ALIAS statement.

SYBASE is case-sensitive. SYBASE uses the exact case of a SQL statement when creating database objects like tables and columns. In a SQL statement, users must use the same exact case when referring to these objects. By default, JAM/DB*i* initializes case-sensitive engines using the DM_PRESERVE_CASE flag. This means that JAM/DB*i* matches the SYBASE column name to a JAM variable with the same name and case.

By changing this flag, you can force JAM/DB*i* to perform case-insensitive searches. Use DB_FORCE_TO_LOWER_CASE to match SYBASE column names to lower case JAM names; use DM_FORCE_TO_UPPER_CASE to match to upper case JAM names.

You may also set an optional flag to change the behavior of JAM/DB*i*'s default error handler. An application may set either of the following:

DM_DEFAULT_DBI_MSG	Set the default error handler to display standard JAM/DB*i* messages for all error messages.
DM_DEFAULT_ENG_MSG	Set the default error handler to display SYBASE error messages instead of JAM/DB*i* error messages.

If neither flag is used, DM_DEFAULT_DBI_MSG is the default. To show SYBASE error messages as the default, use the bitwise OR operator and DM_DEFAULT_ENG_MSG :

```
static vendor_t vendor_list[] =
{
    {"sybase", dm_sybsup, DM_PRESERVE_CASE | DM_DEFAULT_ENG_MSG,
    (char *) 0 },

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

If you modify the settings in dbiinit.c, you must recompile and link the JAM/DB*i* executables, jxdbi and jamdbi. dbiinit.c does not affect the utility executables, tbl2f and f2tbl.

Please note that DM_DEFAULT_DBI_MSG and DM_DEFAULT_ENG_MSG do not affect an application using an error hook function. An error hook function is installed with DBMS ONERROR and controls all error message display.

1.2 CONNECTION

See JAM/DB*i Manual – Section 7.2*

The following options are supported for connections to SYBASE:

`USER`	***user_name***
`PASSWORD`	***password***
`SERVER`	***server_name***
`DATABASE`	***database_name***
`INTERFACES`	***interfaces_file_pathname***
`CURSORS`	***1 / 2***
`TIMEOUT`	***seconds***

Use `INTERFACES` to supply the pathname to an interfaces file. An interfaces file contains the name and network address of every SYBASE server available on the network. If this option is not used, SYBASE looks for a file called `interfaces` in the SYBASE parent directory (e.g., `/usr/sybase/interfaces`). This option is ignored for OS/2, MS-DOS, and Windows applications.

Use `TIMEOUT` to set the number of seconds that Open Client waits for a SYBASE response to a request for a connection. A timeout of 0 seconds represents an infinite timeout period. The default is usually 60 seconds.

Use `CURSORS` to control the number of default cursors JAM/DB*i* creates when the application declares a connection. The default is 1. This means that JAM/DB*i* uses one cursor for any operation executed with `sql` or `dm_sql`, whether it is a `SELECT` or non-`SELECT` operation. The application must set `CURSORS` to 2 to use browse mode. You may also wish to use two default cursors if your application switches between a `SELECT` and non-`SELECT` operations. See the section on cursors for additional information.

The syntax for declaring a connection is,

```
dbms DECLARE connection CONNECTION FOR \
    USER user_name PASSWORD password DATABASE database \
    SERVER server INTERFACES interface_pathname \
    TIMEOUT timeout CURSORS number_of_cursors
```

For example,

```
dbms DECLARE dbi_session CONNECTION FOR \
    USER :uname PASSWORD :pword DATABASE sales \
    SERVER birch INTERFACES '/usr/sybase/interfaces.app'
    TIMEOUT 15 CURSORS 2
```

where `uname` and `pword` are JAM field names.

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

1.3 CURSORS

See JAM/DB*i Manual – Section 7.3*

JAM/DB*i* uses two cursors for operations performed by `sql` and its equivalents, `dm_sql` and `dm_sql_noexp`. JAM/DB*i* uses one cursor for SELECT statements and the other for non-SELECT statements. These two cursors may be sufficient for small applications. Larger applications often require more; an application may declare named cursors using DBMS DECLARE CURSOR. For example, master and detail applications often need to declare at least one named cursor: one cursor selects the master rows and additional cursors select detail rows. In short, if an application is processing a SELECT set in increments (i.e., by using DBMS CONTINUE) while it is executing other SELECT statements, two or more cursors are necessary.

JAM/DB*i* does not put any limit on the number of cursors an application may declare to an SYBASE engine. Since each cursor requires memory and SYBASE resources, however, it is recommended that applications close a cursor when it is no longer needed.

1.4 FORMATTING FOR COLON-PLUS AND BINDING

See JAM/DB*i Manual – Chapter 8*

SYBASE requires a leading dollar sign for values inserted in a `money` column in order to ensure precision. JAM/DB*i* will use a leading dollar sign when it formats `DT_CURRENCY` values. Any other amount formatting characters are stripped. Therefore, if a currency field contained

```
500,000.00
```

JAM/DB*i* would format it as

```
$500000.00
```

1.5 SCROLLING

See JAM/DB*i Manual – Section 9.1.2*

SYBASE has native support for backward scrolling in a SELECT set. Before using any of the following commands

```
dbms [WITH CURSOR cursor] CONTINUE_BOTTOM
dbms [WITH CURSOR cursor] CONTINUE_TOP
dbms [WITH CURSOR cursor] CONTINUE_UP
```

the application must specify whether to use native scrolling or JAM/DB*i* scrolling. To use native scrolling, use the command

```
dbms [WITH CURSOR cursor] SET_BUFFER arg
```

where ***arg*** is the number of rows to buffer.

To use JAM/DB*i* scrolling, use the command

```
dbms [WITH CURSOR cursor] STORE FILE [filename]
```

1.5.1
Locking

JAM/DB*i* SYBASE developers should consider locking issues when building applications that `SELECT` large amounts of data.

When an application executes a `SELECT` that returns many rows, SYBASE may use a "shared lock" to preserve read-consistency. That is, to preserve the state of the selected data, SYBASE may 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 500 or more rows.

As a part of developing and testing an application, JAM/DB*i* developers should monitor SYBASE's behavior by running the SYBASE command `sp_lock` from another terminal when the application executes a `SELECT`. If a `SELECT` executed by a JAM/DB*i* application is holding a lock, the cursor's *spid* will be listed.

Since 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,

- get all the rows in the `SELECT` set, or
- flush pending rows in the `SELECT` set

An application has two ways of getting the entire `SELECT` set:

- create JAM arrays which are large enough to hold the entire `SELECT` set, or
- use `DBMS STORE FILE` and `DBMS CONTINUE_BOTTOM` to buffer all the rows in a temporary file on disk

For example, an application may set up a continuation file before executing a SELECT. Before returning control to the user, the application may execute DBMS CONTINUE_BOTTOM which forces JAM/DB*i* get all the rows from the SELECT set and buffer them in a temporary file. This also forces SYBASE 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 is sets up a continuation file and executes the SELECT. It calls CONTINUE_BOTTOM to force JAM/DB*i* to get all the rows. Finally, it calls 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
   sql SELECT ....
   dbms CONTINUE_BOTTOM
   dbms CONTINUE_TOP
return
```

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

```
proc big_select
   sql SELECT ....
   if @dmretcode != DM_NO_MORE_ROWS
      dbms FLUSH
return
```

To monitor lock information within the application, the application may query SYBASE for the spid 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 SYBASE variable @@spid.

```
# Get the SYBASE spid for a JAM/DBi cursor
# before SELECTing rows.
proc get_spid
parms cursor
vars spid
   if cursor == ""
      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 SYBASE spid.
proc lockstatus
    parms spid4select
    vars lcount
    dbms DECLARE lock_count 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 may get a cursor's spid before executing a `SELECT` for rows. After fetching rows the application may query SYBASE 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. See the example below.

```
proc select
vars cursor_spid locks

    retvar cursor_spid
    jpl get_spid "c1"
    retvar

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

    retvar locks
    jpl lockstatus :cursor_spid
    retvar

    msg emsg "The number of lock(s) is " locks
    return
```

1.6 ERROR AND STATUS INFORMATION

See JAM/DB*i Manual – Section 9.2 and Chapter 13*

In Release 5, JAM/DB*i* uses the global variables described in the following sections to supply error and status information in an application. Note that some global variables may not be used in the current release; however, these variables are reserved for use in other engines and for use in future releases of JAM/DB*i* for SYBASE.

1.6.1 Errors

JAM/DB*i* initializes the following global variables for error code information:

`@dmretcode`	Standard JAM/DB*i* status code.
`@dmretmsg`	Standard JAM/DB*i* status message.
`@dmengerrcode`	SYBASE error code.
`@dmengerrmsg`	SYBASE error message.
`@dmengreturn`	Return code an executed stored procedure.

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

All SYBASE errors with a severity greater than 10 are JAM/DB*i* errors. Otherwise, they are considered warnings.

The easiest way to test for SYBASE errors is with an installed error or exit handler. For example,

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

proc errors
parms stmt engine flag
   if @dmengerrcode == 0
      msg emsg 'JAM/DBi error: " @dmretmsg
   else
      msg emsg "JAM/DBi error: " @dmretmsg " %N" \
      ":engine error is" @dmengerrcode " " @dmengerrmsg
   return 1
```

If you need additional information about SYBASE errors, please consult your SYBASE documentation.

1.6.2
Warnings

JAM/DB*i* initializes the following global variables for warning information:

`@dmengwarncode`	SYBASE warning code.
`@dmengwarnmsg`	SYBASE warning message.

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

You may 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
parms stmt engine flag

if @dmengwarncode
   msg emsg "SYBASE Warning is " @dmengwarnmsg
return
```

1.6.3
Row Information

JAM/DB*i* initializes the following global variables for row information:

`@dmrowcount`	Count of the number of SYBASE rows affected by an operation.
`@dmserial`	Not used in JAM/DB*i* for SYBASE.

SYBASE returns a count of the rows affected by an operation. JAM/DB*i* writes this value to the global variable `@dmrowcount`.

As explained on the manual page for `@dmrowcount`, the value of `@dmrowcount` after a `SELECT` is the number of rows fetched to JAM variables. This number is less than or equal to the total number of rows in the select set. Immediately after an `INSERT`, `UPDATE`, or `DELETE`, `@dmrowcount` is set to the total number of rows affected by the operation. This variable is cleared whenever a `DBMS COMMIT` statement is executed.

The value of `@dmrowcount` may be unexpected after executing a stored procedure. If the stored procedure executes a `SELECT`, `@dmrowcount` equals the number of rows fetched. If, however, the stored procedure does an `INSERT`, `UPDATE`, or `DELETE`, `@dmrowcount` is set to –1. This is documented SYBASE behavior. If you need this information, SYBASE recommends that you test for it within the stored procedure and return it as an output parameter or return code. `@rowcount` is a SYBASE 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
```

See your SYBASE Command Reference Manual for more information.

1.7 UTILITIES

See JAM/DB*i* *Manual – Chapter 16*

If you start the utilities in interactive mode using the `-i` flag, the utility displays an engine-independent logon screen. JAM/DB*i* uses the following options:

- `User`
- `Password`
- `Server name`
- `Database name`

when declaring a connection to SYBASE for the utilities. Enter the same information you use to declare a connection in `jamdbi`. The other fields on the logon screen may remain empty.

1.7.1 f2tbl

`f2tbl` creates a database table based on a JAM form. It uses each named field on the form to create a column, translating field edits to an appropriate SYBASE column definition. The table below shows the default SYBASE column definitions for each JAM type.

If you do not know how to check a field's JAM type, please see the *Utility Reference Chapter* of the JAM/DB*i* manual.

JAM *Type*	*SYBASE Column Definition*		
	Type	*Length*	*Precision*
`DT_CURRENCY`	`money`		
`DT_DATETIME`	`datetime`		
`DT_YESNO`	`char`	Same as field length	
`FT_CHAR`	`char`	Same as field length	
`FT_DOUBLE`	`float`		
`FT_FLOAT`	`float`		
`FT_INT`	`int`		
`FT_LONG`	`int`		
`FT_PACKED`	`float`		
`FT_SHORT`	`smallint`		
`FT_UNSIGNED`	`int`		
`FT_VARCHAR`	`varchar`	Same as field length	
`FT_ZONED`	`float`		

The utility assigns a length for character-type columns. For all other columns, it uses the default length of the datatype.

To change these defaults you must edit the JPL procedure `type` in the distribution JPL module `sybf2t.jpl`, compile it by using `jpl2bin`, and replace the previous version in `sybjpl.lib` by using `formlib -r`.

1.7.2 tbl2f

`tbl2f` creates a JAM form based on an SYBASE table. It creates a field for each column in the table, using the column's datatype to assign the appropriate field characteristics. The table below lists the following for each SYBASE datatype: the identification number for that datatype from the SYBASE system table `systypes`, the default JAM type and the default field length and precision.

JAM/DB*i* by default preserves the same case when creating field names for a `tbl2f` screen. This is consistent with the default case setting for SYBASE in `dbiinit.c` (see Section 1.1). If you changed the default in `dbiinit.c` to DM_FORCE_TO_LOWER_CASE or DM_FORCE_TO_UPPER_CASE, you should set the case option of `tbl2f` to match. The case option may be set on the command line or from a pull-down menu in interactive mode. For example, to start `tbl2f` in interactive mode and use upper case for JAM variables, type

```
tbl2f -i -lu
```

Note that there are additional characteristics associated with each JAM type. Those are described in the *Utility Reference Chapter* of the JAM/DB*i* manual.

SYBASE Type	JAM *Field Definition*		
	JAM *Type*	*Length*	*Precision*
smallint 52	FT_SHORT	6	
tinyint 48	FT_SHORT	3	
timestamp, varbinary 37	FT_UNSIGNED		
int 45	FT_UNSIGNED	11	
bit 50	FT_UNSIGNED	11	
int 56	FT_LONG	11	
intn 38	FT_LONG	11	
float 62	FT_FLOAT	25	5
floatn 59, 109	FT_FLOAT	25	5
char 47	FT_CHAR		
money 60	DT_CURRENCY	11	
moneyn 110, 122	DT_CURRENCY	11	
datetime 58, 61	DT_DATETIME	20	

SYBASE Type	JAM *Field Definition*		
	JAM *Type*	*Length*	*Precision*
`datetimen` 111	DT_DATETIME	20	
`varchar` 35	FT_VARCHAR	255	
`sysname, varchar` 39	FT_VARCHAR		

To change these defaults, or to add other datatypes, you must edit the JPL procedure `type` in the distribution JPL module `sybt2f.jpl`, compile it by using `jpl2bin`, and replace the previous version in `sybjpl.lib` by using `formlib -r`.

1.8 STORED PROCEDURES

An application may execute a stored procedure with the command `sql` and the engine's command for execution. For example,

```
sql EXEC procedure
```

executes the named stored procedure. An application may also use a named cursor to execute a stored procedure.

```
dbms DECLARE cursor CURSOR FOR \
    [declare parameter type [declare parameter type...]]\
    EXEC procedure [ parameter [OUT], [parameter [OUT]...] ]
dbms [WITH CURSOR cursor] EXECUTE [USING values]
```

For example, if `emp_grades` is the following stored procedure,

```
create proc emp_grades @gval char(1)
as
select last, first from emp where grade = @gval
```

either of the following,

```
sql EXEC emp_grades :+grade
```

or

```
dbms DECLARE x CURSOR FOR EXEC emp_grades ::g_parm
dbms WITH CURSOR x EXECUTE USING grade
```

executes the stored procedure, selecting the names of all employees with the specified grade. If the current screen (or LDB) contains the fields `last` and `first`, the procedure writes the values to **JAM**.

Remember, double colons (::) in a DECLARE CURSOR statement are for cursor parameters. A value is supplied for the employee grade each time the cursor is executed. If a single colon or colon-plus were used, the employee grade would be supplied when the cursor was declared, not when it was executed. See Section 8.2 in the **JAM/DB*i*** manual for more information.

If the DBMS supports output parameters, the keyword OUT traps the value of an output parameter in a **JAM** variable. For example, if `summ_by_grade` is the following stored procedure,

```
  create proc summ_by_grade
    @cnt int output, @asal money output, @gr char(1)
  as
  create table empsum (ss char(11), sal money)
 insert into empsum select emp.ss, acc.sal from emp, acc
     where emp.ss=acc.ss and emp.grade = @gr
  select @cnt = count(*) from empsum
  select @asal = avg(sal) from empsum
  drop table empsum
```

the application should declare a cursor for the procedure:

```
dbms DECLARE cur1 CURSOR FOR \
   declare @t1 int declare @t2 money \
   EXEC summ_by_grade @cnt=@t1 OUT, @asal=@t2 OUT, \
   @gr=::grade_parm
dbms WITH CURSOR cur1 EXECUTE USING gr = grade
```

If `cnt` and `asal` are **JAM** variables, the procedure returns the number of employees in the specified grade and their average salary. Note that `t1` and `t2` are temporary SYBASE variables, not **JAM** variables. SYBASE requires that output values be passed as variables, not as constants. The application may use DBMS ALIAS to map the values of output parameters to **JAM** variables. For example,

```
dbms DECLARE cur1 CURSOR FOR \
   declare @t1 int declare @t2 money \
   EXEC summ_by_grade @cnt=@t1 OUT, @asal=@t2 OUT, \
   @gr=::grade_parm
dbms WITH CURSOR cur1 ALIAS cnt emp_count, asal sal_avg
dbms WITH CURSOR cur1 EXECUTE USING gr = grade
```

maps the value of `cnt` to the JAM variable `emp_count` and the value of `asal` to the JAM variable `sal_avg`.

1.8.1
Remote Procedure Calls

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

- One or more of the stored procedure's parameters has a datatype that is not `char`. An rpc is more efficient in these cases because it is capable of passing parameters in their native datatypes 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 an remote procedure call, an application performs the following steps:

1. Must declare an rpc cursor.
2. Must declare the datatype of each parameter that has a non-char datatype.
3. May specify aliases for output parameters or selected columns.
4. Must execute the cursor, supplying in the USING clause a JAM variable for each parameter.

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

Declaring the rpc Cursor

JAM/DB*i* uses binding to support rpcs. 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 JAM/DB*i* 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.

Datatyping the rpc Parameters

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

```
dbms [WITH] CURSOR cursor TYPE [parameter] engine_datatype \
    [, [parameter] engine_datatype ...]
```

parameter is a parameter in the DBMS DECLARE CURSOR statement. ***engine_datatype*** is the datatype of a parameter in the procedure. If parameter names are not given, the types are assigned by position.

JAM/DB*i* 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 performed for binding.

Redirecting the Value of Output Parameter

By default, when an rpc cursor with an output parameter is executed, JAM/DB*i* searches for a JAM variable with the same name as the output parameter. To write the output value to a JAM variable with another name, use the DBMS ALIAS command.

```
dbms [WITH] CURSOR cursor ALIAS [output_parameter] jamvar \
    [, [output_parameter] jamvar ...]
```

If the procedure selects rows, aliases may 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 JAM variable for each parameter. The syntax is the following:

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

JAM/DB*i* passes the name of parameter given in the DBMS DECLARE CURSOR statement, the datatype of the parameter given in the DBMS TYPE statement, and the parameter's value which is the value of ***variable***.

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

Example

If `newsal` is the following stored procedure,

```
create proc newsal
  @ssn char(11), @change float,
  @salary money output, @proposed_sal money output
as
select @salary = (select sal from acc where ssn = @ssn)
select @proposed_sal = @salary * (@change + 1)
```

an rpc would be more efficient than an exec cursor because the procedure has an input parameter with a non-char datatype, and because it returns two output parameters.

The following statement declares an rpc cursor for the stored procedure. Note that the keyword `OUT` follows each of the output parameters.

```
dbms DECLARE cur2 CURSOR FOR RPC newsal ::ssn, ::change,\
   ::salary OUT, ::proposed_sal OUT
```

Before executing the cursor, the application must specify the SYBASE datatypes for the three non-character datatypes.

```
dbms WITH CURSOR cur2 TYPE \
   change float, salary money, proposed_sal money
```

When executing the cursor, the application must provide a **JAM** variable for each parameter. **JAM**/DB*i* passes the name, datatype, and value of the parameters to the procedure. Note that the procedure does not use the input value of the parameters `salary` and `proposed_sal`. **JAM**/DB*i*'s binding mechanism, however, requires a variable in the `USING` clause for each parameter.

```
dbms WITH CURSOR cur2 EXECUTE \
   USING ssn, change, salary, proposed_sal
```

The procedure passes its output, the two salary values, to the **JAM** variables `salary` and `proposed_sal`. To put the output values in the fields `sal1` and `sal2`, execute the following:

```
dbms WITH CURSOR cur2 ALIAS salary sal1, \
   proposed_sal sal2
dbms WITH CURSOR cur2 EXECUTE USING ssn=ssn, \
   change=change, salary=currency, proposed_sal=currency
```

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 **JAM** variables.

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

```
dbms DECLARE cur3 CURSOR FOR \
  declare @x money declare @y money \
  EXEC newsal @ssn = ::ssn, @change = ::change, \
  @salary = @x output, @proposed_sal = @y output

dbms WITH CURSOR cur3 EXECUTE USING ssn=ssn, change=change
```

1.8.2 Controlling the Execution of a Stored Procedure

JAM/DB*i* 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
```

where ***behavior*** is one of the following

```
STOP_AT_FETCH
EXECUTE_ALL
```

If ***behavior*** is STOP_AT_FETCH, JAM/DB*i* 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 may 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 may 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 may 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, JAM/DB*i* executes all statements in the stored procedure without halting. If the procedure selects rows, JAM/DB*i* 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.

1.8.3 Trapping a Return Code from a Stored Procedure

JAM/DB*i* 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 @dmengreturn when @dmretcode = DM_END_OF_PROC. See Appendix B in the JAM/DB*i* manual for the value of DM_END_OF_PROC.

Executing a new sql or 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 may be declared and executed for the procedure,

```
# 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 may test the value of `@dmengreturn` and display a message based on the return status code.

```
proc check_ret
# DM_END_OF_PROC is a constant in the LDB.
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
   jpl check_ret
}
return
```

1.9 TRANSACTIONS

On SYBASE, a transaction controls exactly one cursor. Therefore, in a JAM/DB*i* 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 discussion of the JAM/DB*i* commands for two-phase commit is in Section 1.9.2.

The following events commit a transaction on SYBASE:

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

The following events rollback a transaction on SYBASE:

- executing a `DBMS ROLLBACK`.
- closing the transaction's cursor or connection before the transaction is committed

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

1.9.1 Transaction Control on a Single Cursor

Once a connection has been declared, an application may begin a transaction on the default cursor or on any declared cursor.

SYBASE supports the following transaction commands:

- `DBMS [WITH CONNECTION` ***connection***`] BEGIN`
  `DBMS [WITH CURSOR` ***cursor***`] BEGIN`
  Begin a transaction on a default or named cursor.
- `DBMS [WITH CONNECTION` ***connection***`] SAVE` ***savepoint***
  `DBMS [WITH CURSOR` ***cursor***`] SAVE` ***savepoint***
  Create a savepoint in the transaction on a default or named cursor.
- `DBMS [WITH CONNECTION` ***connection***`] COMMIT`
  `DBMS [WITH CURSOR` ***cursor***`] COMMIT`
  Commit the transaction on a default or named cursor.
- `DBMS [WITH CONNECTION` ***connection***`] ROLLBACK [`***savepoint***`]`
  `DBMS [WITH CURSOR` ***cursor***`] ROLLBACK [`***savepoint***`]`
  Rollback to a savepoint or to the beginning of the transaction on a default or named cursor.

A transaction on a default cursor controls all inserts, updates, and deletes executed with the JPL command `sql` or `dm_sql`. The application may set the default connection before beginning the transaction or it may use the `WITH CONNECTION` clause in each statement. A simple transaction on a default cursor may appear as

```
dbms CONNECTION connection
dbms BEGIN
sql statement
sql statement
...
dbms SAVE savepoint
sql statement
dbms ROLLBACK savepoint
dbms COMMIT
```

If a named cursor is declared for multiple statements, it may be useful to execute the cursor in a transaction. This way the application may ensure that SYBASE executes either all of the cursor's statements or none of the cursor's statements. A simple transaction on a named cursor may appear as

```
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 may be executed more than once in the transaction. The application should not, however, redeclare a cursor within a transaction.

Examples are shown below with error handlers.

Example 1. A Transaction on the Default Cursor

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

```
proc tran_handle
{
   parms subroutine
   vars jpl_retcode
   retvar jpl_retcode
# Call the subroutine.
      jpl :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, JAM 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_employee
dbms BEGIN
    sql INSERT INTO emp VALUES \
        (:+ssn, :+last, :+first, \
        :+street, :+city, :+st, :+zip)
    sql INSERT INTO review VALUES \
        (:+ssn, :+startdate, :+startsal, :+grade)
    sql INSERT INTO acc VALUES (:+ssn, :+startsal, :+exmp)
dbms COMMIT
return 0
```

The procedure `tran_handle` is a generic handler for the application's transactions. It is like the one described in the *Developer's Guide*. The procedure `new_employee` contains the transaction statements. This method reduces the amount of error checking code.

The application executes the transaction by executing

```
jpl tran_handle new_employee
```

The procedure `tran_handle` receives the argument "new_employee" and writes it to the variable `subroutine`. It defines and declares a JPL variable to receive a JPL return code. After performing colon processing `:subroutine` is replaced with its value, `new_employee`, and JPL calls the procedure. The procedure `new_employee` begins the transaction, performs three inserts, and commits the transaction.

If `new_employee` 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_employee`, JAM/DB*i* 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_employee` executes successfully but the second `INSERT` fails because it violates the rule `grade_range`. In this case, JAM/DB*i* calls the error handler to display an error message. When the error handler returns the abort code 1, JAM aborts the procedure `new_employee` (therefore, the third `INSERT` is not attempted). JAM 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 `emp`.

1.9.2 Transaction Control on Multiple Cursors

SYBASE 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 servers.

The main transaction must be declared with the command

```
dbms [WITH CONNECTION connection] \
    DECLARE transaction 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. Since SYBASE 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; SYBASE does not permit periods (`.`) or colons (`;`) in a transaction name. Since "transaction" and "tran" are keywords for both JAM/DB*i* and SYBASE, do not use these words for this argument.

- ***application***: the name of the application; it may 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 SYBASE commit and recovery systems and must be set appropriately.

Once the two-phase commit transaction is declared, its name is used to begin and to commit or rollback the transaction. The syntax is

`dbms BEGIN` ***transaction***

`dbms COMMIT` ***transaction***

`dbms ROLLBACK` ***transaction***

As with cursors and connections, JAM/DB*i* 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, JAM/DB*i* calls the support routine to close the connection with the SYBASE commit service. The command is the following:

`dbms CLOSE TRANSACTION` ***transaction***

Operations on a single cursor are subtransactions. To control a subtransaction in a two-phase commit transaction, the following commands may 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. Once the application has "prepared" all the subtransactions, it issues a `COMMIT` to the main transaction and each subtransaction.

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

1. Declare any necessary connections and cursors.
2. Declare the main transaction.

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

3. Begin the main transaction.

   ```
   dbms BEGIN tname
   ```

4. 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
   sql statement
   sql statement
   dbms SAVE savepoint
   sql statement
   dbms ROLLBACK savepoint

   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
   ```

5. Commit the main transaction.

   ```
   dbms COMMIT tname
   ```

6. Commit each subtransaction indicating a named or default cursor.

   ```
   dbms WITH CURSOR cursor1 COMMIT
   dbms COMMIT
   dbms WITH CURSOR cursor2 COMMIT
   ```

7. 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
call 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
jpl 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 var)
dbms BEGIN new_emp

   dbms WITH CURSOR x BEGIN
   cat tran_level "1"
   dbms WITH CURSOR x EXECUTE USING \
      (ss, last, first, street, city, st, zip, grade)
```

```
    dbms WITH CURSOR y BEGIN
    cat 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."
cat tran_level ""
return

########################################################
proc tran_error
vars fail_area [2](20), tran_err(3)
cat fail_area[1] "address"
cat 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
    cat tran_level ""
    return :tranerr
}
msg emsg @dmengerrmsg
return 1
```

1.10 SYBASE-SPECIFIC COMMANDS

See JAM/DB*i* *Manual – Chapter 11*

JAM/DB*i* for SYBASE provides additional commands for SYBASE-specific features. 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.

1.10.1 Using Browse Mode

`BROWSE`	execute a `SELECT` for browsing
`UPDATE`	update a table while browsing

1.10.2 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

1.10.3 Using Transactions

JAM/DB*i* supports the following commands when using transactions. See the reference pages for more information on each command.

`BEGIN`	begin a transaction
`CLOSE_ALL_TRANSACTIONS`	close all transactions declared for two-phase commit
`CLOSE TRANSACTION`	close a named transaction
`COMMIT`	commit a transaction
`DECLARE TRANSACTION`	declare a transaction for two-phase commit
`PREPARE_COMMIT`	prepare to commit a transaction

`ROLLBACK`	rollback a transaction
`SAVE`	save a two-phase commit

BEGIN

start a transaction

SYNOPSIS

```
dbms [WITH CONNECTION connection] BEGIN
dbms [WITH CURSOR cursor] BEGIN

dbms  BEGIN two_phase_commit
```

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 which may involve multiple cursors on different 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. Once a transaction is begun, changes to the database are not committed until a `DBMS COMMIT` is executed. Changes are undone by executing `DBMS ROLLBACK`.

If a `WITH CURSOR` clause is used in a `DBMS BEGIN` statement, JAM/DB*i* begins a transaction on the named cursor. If a `WITH CONNECTION` clause is used, JAM/DB*i* begins a transaction on the default cursor of the named connection. If no `WITH` clause is used, JAM/DB*i* begins a transaction on the default cursor of the default connection.

To begin a distributed transaction (two-phase transaction), first declare a named transaction with `DBMS DECLARE TRANSACTION`. Since this statement supports a `WITH CONNECTION` clause, JAM/DB*i* 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` ***two_phase_commit*** where ***two_phase_commit*** is the name of the declared transaction, JAM/DB*i* 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 JAM/DB*i* will not close a connection with a pending two-phase commit transaction.

SEE ALSO

Section 1.9 – Transactions.

Documentation provided by the database vendor.

RELATED COMMANDS

`dbms COMMIT`

`dbms ROLLBACK`

`dbms SAVE`

EXAMPLE

Refer to the examples in *Section 1.9 – Transactions.*

BROWSE

retrieve SELECT results one row at a time

SYNOPSIS

dbms BROWSE ***SELECTstmt***

DESCRIPTION

This command allows an application to execute a SELECT in "browse" mode. This means that SYBASE will return the SELECT rows one at a time to the JAM/DB*i* application; SYBASE will not set any shared locks for the SELECT. The application may use the companion command DBMS UPDATE to update the current row. SYBASE will verify that the row has not been changed before it issues the UPDATE.

To use 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 may 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. JAM/DB*i* 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. It the table is not browsable, JAM/DB*i* returns the DM_BAD_ARGS error. If a table is browsable, JAM/DB*i* returns the first row in the select set when DBMS BROWSE is executed. Note that only row is returned at a time.

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

RELATED COMMANDS

dbms CONTINUE

dbms FLUSH

dbms UPDATE

EXAMPLE

```
# Browse mode requires a connection declared with 2
# cursors.
dbms DECLARE browse_con CONNECTION FOR \
   USER :user PASSWORD :pass SERVER :server CURSORS 2
```

```
proc start_browse_mode
   dbms CONNECTION browse_con
   dbms BROWSE SELECT ss, last, first, sal FROM employee
   return

proc update_browse_row
# Allow the user to update the employee salary. DBi builds
# the WHERE clause to identify this row.
   dbms UPDATE employee SET sal = :+sal
   return

proc next_browse_row
# Fetch the next row.
   dbms CONTINUE
   return
```

CANCEL

cancel the execution of a stored procedure

SYNOPSIS

```
dbms [WITH CURSOR cursor] CANCEL
```

DESCRIPTION

This command cancels any outstanding work on the named cursor. In particular, this command may be used to cancel a pending stored procedure. 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

If the `WITH CURSOR` clause is not used, JAM/DB*i* executes the command on the default cursor.

SEE ALSO

Section 1.8 – Stored Procedures.

RELATED COMMANDS

```
dbms FLUSH
```

CLOSE_ALL_TRANSACTIONS

close all transactions declared for two-phase commit

SYNOPSIS

```
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`, JAM/DB*i* will return the error `DM_TRAN_PENDING`.

If an application terminates with a pending two-phase commit transaction, SYBASE will mark the transaction's process as "infected." You will need the system administrator to delete the infected process. To help prevent this, JAM/DB*i* will not close a connection unless all two-phase commit transactions have been closed. Furthermore, JAM/DB*i* will not close a two-phase commit transaction unless the application explicitly terminated the transaction with a `DBMS COMMIT` ***two_phase_commit*** or `DBMS ROLLBACK` ***two_phase_commit***.

Since this command verifies that all two-phase commit transactions were terminated, you may wish to call this command before logging off.

SEE ALSO

Section 1.9 – Transactions.

RELATED COMMANDS

```
dbms BEGIN

dbms CLOSE TRANSACTION

dbms COMMIT

dbms DECLARE TRANSACTION

dbms ROLLBACK
```

EXAMPLE

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

```
# APP1 = ^jpl two_phase_cleanup
proc cleanup_failure
   parms stmt engine flag
   if @dmretcode == DM_TRAN_PENDING
   {
      call jm_keys APP1
   }
   return 0


proc two_phase_cleanup
   dbms ROLLBACK ...
   dbms CLOSE TRANSACTION ...
   return
```

CLOSE TRANSACTION

close a declared transaction structure

SYNOPSIS

```
dbms CLOSE TRANSACTION transaction
```

DESCRIPTION

This command closes the main transaction which 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 SYBASE 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

Section 1.9 – Transactions.

RELATED COMMANDS

```
dbms BEGIN
dbms COMMIT
dbms DECLARE TRANSACTION
dbms PREPARE_COMMIT
dbms ROLLBACK
dbms SAVE
```

COMMIT

commit a transaction

SYNOPSIS

```
dbms [WITH CONNECTION connection] COMMIT
dbms [WITH CURSOR cursor] COMMIT
dbms COMMIT two_phase_commit
```

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 `DBMS 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, JAM/DB*i* commits the transaction on the named cursor. If a `WITH CONNECTION` clause is used, JAM/DB*i* commits the transaction on the default cursor of the named connection. If no `WITH` clause or no distributed transaction name is used, JAM/DB*i* commits the transaction on the default cursor of the default connection.

If a distributed transaction name is used, JAM/DB*i* 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.

SEE ALSO

Section 1.9 – Transactions.

RELATED COMMANDS

```
dbms BEGIN
dbms CLOSE TRANSACTION
dbms DECLARE TRANSACTION
dbms PREPARE_COMMIT
dbms ROLLBACK
dbms SAVE
```

EXAMPLE

Refer to the example in *Section 1.9 – Transactions.*

DECLARE CURSOR FOR RPC

declare a named cursor for a remote procedure

SYNOPSIS

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

DESCRIPTION

Use this command to create or redeclare a named cursor to execute a remote procedure call (rpc). Since JAM/DB*i* 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 JAM/DB*i* 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 datatype may be given in the `DBMS DECLARE CURSOR` statement, or in a `DBMS TYPE` statement.

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

SEE ALSO

Section 1.8 – Stored Procedures.

`@dmengreturn`

RELATED COMMANDS

`dbms CLOSE CURSOR`

`dbms WITH CURSOR` ***cursor*** `EXECUTE`

`dbms TYPE`

`WITH CURSOR`

EXAMPLE

Refer to the example in *Section 1.8 – Stored Procedures.*

DECLARE TRANSACTION

declare a named transaction for two phase commit

SYNOPSIS

```
dbms [WITH CONNECTION connection] \
    DECLARE transaction TRANSACTION FOR \
    SITES sites APPLICATION application
```

DESCRIPTION

This command declares a two-phase commit transaction structure.

The `WITH CONNECTION` clause identifies the server which 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.

transaction is the name of the two-phase commit transaction. Do not use the keywords "tran" or "transaction" for this argument. The application will use this name to begin, to commit or rollback, and to close the transaction.

sites is 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.

application is an optional argument which identifies the name of the transaction.

The application must use ***transaction*** to begin and commit or rollback the two-phase commit.

After declaring the transaction, begin the transaction using `DBMS BEGIN`. When the transaction is complete, close the transaction using either `CLOSE TRANSACTION` or `CLOSE_ALL_TRANSACTIONS`. An application must close all declared transactions before closing their connections.

SEE ALSO

Section 1.9 – Transactions.

RELATED COMMANDS

```
dbms CLOSE TRANSACTION transaction
```

EXAMPLE

Refer to the examples in *Section 1.9 – Transactions.*

FLUSH

flush any selected rows not fetched to JAM variables

SYNOPSIS

```
dbms [WITH CURSOR cursor] FLUSH
```

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 may execute `DBMS CONTINUE` until the `DM_NO_MORE_ROWS` signal is returned, or it may execute `DBMS FLUSH` which cancels the pending rows.

This command is also useful with queries that fetch very large `SELECT` sets. The application may 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, once the rows have been flushed, the application may not use `DBMS CONTINUE` to view the unread rows.

RELATED COMMANDS

```
dbms DECLARE CURSOR

dbms CANCEL

dbms CONTINUE

dbms NEXT
```

EXAMPLE

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

NEXT

execute the next statement in a stored procedure

SYNOPSIS

```
dbms [WITH CURSOR cursor] NEXT
```

DESCRIPTION

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

SEE ALSO

Section 1.8 – Stored Procedures.

RELATED COMMANDS

```
dbms DECLARE CURSOR
dbms CANCEL
dbms CONTINUE
dbms FLUSH
dbms SET [EXECUTE_ALL | SINGLE_STEP | STOP_AT_FETCH ]
```

EXAMPLE

Refer to the example in *Section 1.8 – Stored Procedures.*

PREPARE_COMMIT

prepare a two phase commit

SYNOPSIS

```
dbms [WITH CURSOR cursor] PREPARE_COMMIT
```

DESCRIPTION

Use of this command is required during the two-phase commit service. It needs to be 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. Once 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, JAM/DB*i* issues the command on the default cursor.

SEE ALSO

Section 1.9 – Transactions

RELATED COMMANDS

```
dbms BEGIN
dbms CLOSE TRANSACTION
dbms COMMIT
dbms DECLARE TRANSACTION
dbms ROLLBACK
dbms SAVE
```

EXAMPLE

Refer to the example in *Section 1.9 – Transactions.*

ROLLBACK

rollback a transaction

SYNOPSIS

```
dbms [WITH CONNECTION connection] ROLLBACK savepoint
dbms [WITH CURSOR cursor] ROLLBACK savepoint

dbms ROLLBACK two_phase_commit
```

DESCRIPTION

Use this command to rollback a transaction and restore the database to its state prior to the start of the transaction.

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, JAM/DB*i* rollbacks the transaction on the named cursor. If a `WITH CONNECTION` clause is used, JAM/DB*i* rollbacks the transaction on the default cursor of the named connection. If no `WITH` clause or no distributed transaction name is used, JAM/DB*i* rollbacks the transaction on the default cursor of the default connection.

If a distributed transaction name is used, JAM/DB*i* 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

Section 1.9 – Transactions

RELATED COMMANDS

```
dbms BEGIN
dbms COMMIT
dbms DECLARE TRANSACTION
dbms PREPARE_COMMIT
dbms ROLLBACK
dbms SAVE
```

EXAMPLE

Refer to the example in *Section 1.9 – Transactions.*

SAVE

set a savepoint or checkpoint within a transaction

SYNOPSIS

```
dbms [WITH CONNECTION connection] SAVE savepoint
dbms [WITH CURSOR cursor] SAVE savepoint
```

DESCRIPTION

This command creates a savepoint in the transaction. A savepoint is a pointer set by the programmer within a transaction. When a savepoint is set, the procedures following the savepoint can be cancelled using `DBMS ROLLBACK` ***savepoint***.

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

SEE ALSO

Section 1.9 – Transactions

RELATED COMMANDS

```
dbms BEGIN
dbms COMMIT
dbms DECLARE TRANSACTION
dbms PREPARE_COMMIT
dbms ROLLBACK
dbms SAVE
```

EXAMPLE

Refer to the example in *Section 1.9 – Transactions.*

SET

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

SYNOPSIS

```
dbms [WITH CURSOR cursor] SET \
   [EXECUTE_ALL | SINGLE_STEP | STOP_AT_FETCH]
```

DESCRIPTION

This command controls the execution of a stored procedure or a cursor with multiple statements. Its options are

EXECUTE_ALL	Specifies that the DBMS return control to JAM/DB*i* only when all statements have been executed or when an error occurs. If a SELECT is executed, only the first pageful of rows is returned to JAM variables. This option may be set for a multi-statement or a stored procedure cursor.
SINGLE_STEP	Specifies that the DBMS return control to the JAM Executive after executing each statement belonging to the multi-statement cursor. After each SELECT, the user may 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 may be set for a multi-statement cursor. If this option is used with a stored procedure cursor, JAM/DB*i* uses the default setting STOP_AT_FETCH.
STOP_AT_FETCH	Specifies that the DBMS return control to the JAM Executive after executing a SELECT that fetches rows. (Note that control is not returned for a SELECT that assigns a value to a local SYBASE parameter.) The application may 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 may 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.

SEE ALSO

Section 1.8 – Stored Procedures

RELATED COMMANDS

```
dbms CANCEL

dbms CONTINUE

dbms DECLARE CURSOR

dbms DECLARE CURSOR FOR EXEC

dbms DECLARE CURSOR FOR RPC

dbms FLUSH

dbms NEXT
```

EXAMPLE

```
vars DM_NO_MORE_ROWS(5) DM_END_OF_PROC(5)
cat DM_NO_MORE_ROWS "53256"
cat DM_END_OF_PROC "53270"

dbms DECLARE x CURSOR FOR \
   SELECT company, street, city, st, zip \
      FROM client_list WHERE co_id = ::company_id \
   INSERT INTO contacts VALUES \
      (::newfirst, ::newlast, ::newloc, ::newphone) \
   SELECT first, last, location, phone FROM contacts \
      WHERE co_id = ::company_id
msg d_msg "%KPF1 START  %KPF2 SCROLL SELECT\
  %KPF3 EXECUTE NEXT STEP"

proc f1
dbms WITH CURSOR x SET SINGLE_STEP
dbms WITH CURSOR x EXECUTE USING company_id, newfirst, \
   newlast, newloc, newphone, company_id
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
```

SET_BUFFER

set up a buffer for engine-supported scrolling

SYNOPSIS

```
dbms [WITH CURSOR cursor] SET_BUFFER [number_of_rows]
```

DESCRIPTION

SYBASE supports non-sequential scrolling if the application has set up a buffer for result rows. If an application does not need `DBMS CONTINUE_UP` or is using a continuation file (`DBMS STORE FILE`), this command is not needed.

number_of_rows is the number of rows SYBASE will buffer. To be useful, ***number_of_rows*** should be greater than the number of occurrences in the JAM destination fields.

When this command is used with a `SELECT` cursor, SYBASE saves the specified number of result rows of the `SELECT` 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 the command,

```
dbms [WITH CURSOR cursor] 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 redeclaring 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.

SEE ALSO

```
dbms STORE [FILE [filename]]
```

RELATED COMMANDS

```
dbms CONTINUE_BOTTOM
dbms CONTINUE_TOP
dbms CONTINUE_UP
```

EXAMPLE

```
dbms DECLARE emp_cursor CURSOR FOR SELECT * FROM emp
dbms WITH CURSOR emp_cursor SET_BUFFER 500

proc scroll_up
dbms WITH CURSOR emp_cursor CONTINUE_UP
return

proc scroll_down
dbms WITH CURSOR emp_cursor CONTINUE_DOWN
return
```

TRANSACTION

set a default declared two-phase commit transaction

SYNOPSIS

`dbms TRANSACTION` ***variable***

DESCRIPTION

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

RELATED COMMANDS

```
dbms BEGIN
dbms COMMIT
dbms DECLARE TRANSACTION
dbms PREPARE_COMMIT
dbms ROLLBACK
dbms SAVE
```

TYPE

declare parameter datatypes for an rpc cursor

SYNOPSIS

```
dbms WITH CURSOR cursor TYPE parameter datatype \
    [, parameter datatype ...]
```

DESCRIPTION

If an application has declared a cursor for a remote procedure call ("rpc") but has not declared the datatypes 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. ***datatype*** is the datatype of the parameter in the stored procedure. JAM/DB*i* uses the information supplied with this command to execute the remote procedure call. Please note that these datatypes 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.

SEE ALSO

Section 1.8 – Stored Procedures

RELATED COMMANDS

```
dbms DECLARE cursor CURSOR FOR RPC procedure \
  [::parameter [OUT] datatype [, ::parameter [OUT] datatype ...]

dbms DECLARE cursor CURSOR FOR RPC procedure \
  [::parameter [OUT] [, ::parameter [OUT] ...]
```

EXAMPLE

```
#######################################################
#procedure newsal:
#create proc newsal @ss char(11), @change float,
# @salary money output, @proposed_sal money output
# as
#  select @salary = (select sal from acc where ss = @ss)
#  select @proposed_sal = @salary * (@change + 1)
#######################################################
```

```
dbms DECLARE sal_cursor CURSOR FOR \
   RPC newsal ::ss, ::change, ::salary OUT, \
    ::proposed_sal OUT

dbms WITH CURSOR sal_cursor TYPE \
   change float, salary money, proposed_sal money

dbms WITH CURSOR sal_cursor EXECUTE \
   USING ss, change, salary, proposed_sal
```

UPDATE
update a table while browsing

SYNOPSIS

```
dbms UPDATE table 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 two other procedures to execute `DBMS CONTINUE` and `DBMS UPDATE`.

Please note that the `DBMS UPDATE` statement has no `WHERE` clause. JAM/DB*i* calls a SYBASE 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 SYBASE will not perform the update.

RELATED COMMANDS

```
dbms BROWSE
dbms CANCEL
dbms CONTINUE
dbms FLUSH
```

EXAMPLE

See manual page for `DBMS BROWSE`.

USE
open an existing database

SYNOPSIS

```
dbms [WITH CONNECTION connection] USE database
```

DESCRIPTION

This command changes a connection's default database. ***database*** must be an existing database, and the user must have the appropriate permissions to use the database or else **JAM/DB*i*** returns an error.

RELATED COMMANDS

```
dbms DECLARE connection CONNECTION FOR [USER user [PASSWORD
password]] [SERVER server] [DATABASE database] [CURSORS [1|2]]
[INTERFACES filename] [TIMEOUT seconds]
```

EXAMPLE

```
dbms DECLARE c1 CONNECTION FOR \
   USER :uname PASSWORD :pword SERVER :server \
   DATABASE master
sql SELECT * FROM emp
dbms WITH CONNECTION c1 USE projects
sql SELECT * FROM newjobs
```

1.11 COMMAND DIRECTORY FOR SYBASE

This section contains a directory for all the commands available in JAM/DB*i* for SYBASE. The following table lists the command, a short description of the command, and the location of the reference page for that command. If the location is described as SYBASE Notes, that information is enclosed in this document.

Command	*Description*	*Documentation*
`ALIAS`	name a JAM variable as the destination of a selected column or aggregate function	JAM/DB*i* Manual
`BEGIN`	begin a transaction	SYBASE Notes
`BINARY`	create a JAM/DB*i* variable for fetching binary values	JAM/DB*i* Manual
`BROWSE`	execute a `SELECT` for browsing	SYBASE Notes
`CANCEL`	abort execution of a stored procedure	SYBASE Notes
`CATQUERY`	redirect `SELECT` results to a file or a JAM variable	JAM/DB*i* Manual
`CLOSE_ALL_CONNECTIONS`	close all connections on all engines	JAM/DB*i* Manual
`CLOSE_ALL_TRANSACTIONS`	close all transactions	SYBASE Notes
`CLOSE CONNECTION`	close a named connection	JAM/DB*i* Manual
`CLOSE CURSOR`	close a cursor	JAM/DB*i* Manual
`CLOSE TRANSACTION`	close a named transaction	SYBASE Notes
`COMMIT`	commit a transaction	SYBASE Notes
`CONNECTION`	set a default connection and engine for the application	JAM/DB*i* Manual

Command	*Description*	*Documentation*
`CONTINUE`	fetch the next screenful of rows from a `SELECT` set	JAM/DB*i* Manual
`CONTINUE_BOTTOM`	fetch the last screenful of rows from a `SELECT` set	JAM/DB*i* Manual
`CONTINUE_DOWN`	fetch the next screenful of rows from a `SELECT` set	JAM/DB*i* Manual
`CONTINUE_TOP`	fetch the first screenful of rows from a `SELECT` set	JAM/DB*i* Manual
`CONTINUE_UP`	fetch the previous screenful of rows from a `SELECT` set	JAM/DB*i* Manual
`DECLARE CONNECTION`	declare a named connection to an engine	JAM/DB*i* Manual
`DECLARE CURSOR`	declare a named cursor	JAM/DB*i* Manual
`DECLARE CURSOR FOR RPC`	declare a cursor to execute a stored procedure using a remote procedure call	SYBASE Notes
`DECLARE TRANSACTION`	declare a transaction for two-phase commit	SYBASE Notes
`ENGINE`	set the default engine for the application	JAM/DB*i* Manual
`EXECUTE`	execute a named cursor	JAM/DB*i* Manual
`FLUSH`	abort execution of a stored procedure	SYBASE Notes
`FORMAT`	format the results of a `CATQUERY`	JAM/DB*i* Manual
`NEXT`	execute the next statement in a stored procedure	SYBASE Notes

Command	*Description*	*Documentation*
`OCCUR`	set the number of rows for JAM/DB*i* to fetch to an array and choose an occurrence where JAM/DB*i* should begin writing result rows	JAM/DB*i* Manual
`ONENTRY`	install a JPL procedure or C function which JAM/DB*i* will call before executing a `sql` or dbms statement	JAM/DB*i* Manual
`ONERROR`	install a JPL procedure or C function which JAM/DB*i* will call whenever a `sql` or dbms statement fails	JAM/DB*i* Manual
`ONEXIT`	install a JPL procedure or C function which JAM/DB*i* will call after executing a `sql` or dbms statement	JAM/DB*i* Manual
`PREPARE_COMMIT`	prepare to commit a transaction	SYBASE Notes
`ROLLBACK`	rollback a transaction	SYBASE Notes
`SAVE`	save a two-phase commit	SYBASE Notes
`SET`	set execution behavior for a procedure (execute all, stop at fetch, etc.)	SYBASE Notes
`SET_BUFFER`	set up a buffer for engine–supported scrolling	SYBASE Notes
`START`	set the first row for JAM/DB*i* to return from a `SELECT` set	JAM/DB*i* Manual
`STORE`	store the rows of a `SELECT` set in a temporary file so that the application may scroll through the rows	JAM/DB*i* Manual
`TRANSACTION`	set the default transaction	SYBASE Notes
`TYPE`	set data types for parameters of a stored procedure executed with an rpc cursor	SYBASE Notes

Command	*Description*	*Documentation*
`UNIQUE`	suppress repeating values in a selected column	JAM/DB*i* Manual
`UPDATE`	update a table while browsing	SYBASE Notes
`USE`	open an existing database	SYBASE Notes
`WITH CONNECTION`	set the default connection for the duration of a command	JAM/DB*i* Manual
`WITH CURSOR`	specify the cursor to use for a statement	JAM/DB*i* Manual
`WITH ENGINE`	set the default engine for the duration of a command	JAM/DB*i* Manual