New Features in JAM/DBi for SYBASE / SQL Server Release 4.8

Contents

I. Multiple Cursors 1
Multiple Cursors and SELECT Statements 2
Cursor Management 3
Transactions 3

II. Error Processing 4
Error Types 5
End-of-SELECT Signal 5
DBMS START and Error Signalling 6

III. Text Datatype and Word-Wrapped Arrays 7
SELECTing into Word-Wrapped Arrays 7
Updating from a Word-Wrapped Array 7

IV. Customizing Query Result Destinations 8
DBMS Redirect 9
DBMS Catquery 10
DBMS Occur 10

V. Miscellaneous 12
Suppressing Repeating Values in a Query 12
Printing a File 13
JAM Variables and JPL Variables 14
NULL Values 14
Scrolling Through the SELECTed Rows 15
Selecting BINARY Datatypes 17
Browse Mode 17
Handling Stored Procedures and Their Results 18

I. Multiple Cursors

In the following,

cursor_name	is an identifier, consisting of non-blank characters, with maximum same as a JAM variable.
database_name	is the name of the database that the cursor is logging into
sql_statement	is an SQL statement, possibly containing variables (syntax is database specific).

The JAM/DB*i* statements associated with cursors are:

DBMS CONNECT *cursor_name* TO DATABASE *database_name*

This command activates the cursor. It must be issued *before* any other commands which refer to the cursor.

DBMS DECLARE *cursor_name* CURSOR FOR *sql_statement*

This command defines the cursor. It cannot be issued unless *cursor_name* is active (i.e., DBMS CONNECT *cursor_name*... has been issued and DBMS CLOSE CURSOR *cursor_name* has *not* been issued).

DBMS EXECUTE *cursor_name*

This command cannot be issued unless *cursor_name* is active and defined (i.e., CONNECT *cursor_name* and DECLARE *cursor_name* have been issued; CLOSE CURSOR *cursor_name* has *not* been issued).

DBMS CONTINUE [*cursor_name*]

This command continues a preceding EXECUTE. It cannot be issued for a named cursor unless the *cursor_name* is active, defined, and has been executed. CONTINUE *cursor_name* can be re-issued any number of times, so long as DBMS CLOSE CURSOR *cursor_name* has not been issued. A CONTINUE without a cursor name tries to continue the last non-cursor SELECT.

DBMS CLOSE CURSOR *cursor_name*

This command inactivates the named cursor.

Multiple Cursors and SELECT Statements

A SELECT statement may also be associated with a cursor. The additional advantage here is that with two SELECTs associated with two different cursors, the user can jump back and forth without having to re-issue the queries. For example, the following is a valid sequence of JPL statements:

```
DBMS DECLARE sel_nba CURSOR FOR SELECT * FROM NBATEAMS
DBMS DECLARE sel_nfl CURSOR FOR SELECT * FROM NFLTEAMS
DBMS EXECUTE sel_nba      (fetches 16 rows into form NBA)
DBMS EXECUTE sel_nfl      (fetches 16 rows into form NFL)
DBMS CONTINUE sel_nba     (fetches next 16 NBA rows)
DBMS CONTINUE sel_nfl     (fetches next 16 NFL rows)
```

Cursor Management

There may be at most 9 cursors active at any time. This does not include the default cursors (see below). A cursor is active if it has been connected (DBMS CONNECT) and has not been closed (DBMS CLOSE CURSOR). A cursor remains associated with a particular SQL statement until it is either closed, in which case the cursor name ceases to exist, or it is redeclared with another SQL statement. Closing a cursor frees some memory, so it may be useful to keep a minimum number of cursors active.

Upward Compatibility

All non-cursored JAM/DB*i* commands still behave as they used to. Ordinary SQL and DBMS statements may be freely mixed with the cursor commands. Non-cursor JAM/DB*i* commands do not allow arguments, and so may be a little quicker to execute. Other non-SELECT statements or cursored statements may be executed between a non-cursor SELECT and its CONTINUE. The CONTINUE will still try to fetch the next set of rows. For this reason, non-cursor statements may be thought of as using two default cursors, one for SELECT statements and one for non-SELECT statements. The only difference is that arguments are not allowed.

Transactions

Within a transaction, any changes to a table will lock all or portions of that table. This will prevent any other cursor (i.e., other than the one associated with the transaction) from accessing the table. For instance, using just non-cursor commands inside a transaction, a SELECT will not be able to retrieve rows from a table being modified in that transaction.

Execution of select and non-select statements without named cursors (i.e., using the default cursor) actually activates two independent cursors. Therefore, any transaction which includes both select and non-select statements should use named cursors.

II. Error Processing

There are 2 DBMS statements for handling errors:

DBMS ERROR_CONTINUE

DBMS ERROR [*number_var* [*message_var*]]

where:

number_var, *message_var* — are **JAM** variable or field identifiers. They may be array element identifiers, with one of the following form:

id or *id[int]* or *id[id]*

where *id* is a **JAM** identifier, and *int* is an integer.

The default action on any error, either **JAM/DB*i*** or SQL, is to display an error message, followed by the JPL statement that caused the error. When the two messages are acknowledged by hitting the space bar, **JAM/DB*i*** aborts the JPL procedure in which the error occurred. This conforms to the old behavior of version 3.16

Issuing a DBMS ERROR_CONTINUE will prevent **JAM/DB*i*** from aborting the JPL procedure on an error. Error messages will still be displayed as above.

Executing a DBMS ERROR with 1 or 2 **JAM** variable names will cause **JAM/DB*i*** to store the error number and message (if applicable) in the corresponding variables, after which **JAM/DB*i*** moves on to the next statement. A DBMS ERROR without any following variable names causes **JAM/DB*i*** to revert to default behavior.

Note that in the default mode only negative error codes, or those signalling actual errors, are displayed. However when error trapping is on, all errors and informational codes returned by the database are inserted into the appropriate JAM variables.

All internal (JAM/DB*i* or JAM) errors are always displayed at the bottom of the screen, followed whenever possible by the JPL statement during which the error occurred.

Error Types

There are 3 types of errors that can occur while running JAM/DB*i*: SQL errors, JAM/DB*i* errors, and internal errors.

SQL Errors are errors reported by the database system. These usually indicate an error in the SQL statement or database access. SQL errors are displayed at the bottom of the screen by default. This behavior can be modified by the DBMS commands described above. An attempt is made to distinguish between actual *errors* and other informational messages. In most databases, errors have negative numeric codes and informational messages have positive numbers. Only *errors* are displayed on the screen by the default error handler. However error trapping will trap all errors and messages.

JAM/DB*i* *Errors* are errors in the JAM/DB*i* commands that are detected by JAM/DB*i*. An example is an attempt to use an undeclared cursor (see Chapter I). These errors always result in the JPL procedure being aborted and the error message and offending statement being displayed on the screen. Error trapping will not modify this behavior. All such errors should have been caught at application development time.

Internal Errors are errors usually caused by an internal inconsistency in JAM/DB*i*. JAM/DB*i* handles these the same way it handles JAM/DB*i* errors.

End-of-SELECT Signal

An end-of-SELECT signal is not displayed if default error processing is on. However, if error trapping is on, an error code of 100, severity 0 will be trapped in the specified variable. For example:

```
DBMS ERROR errcode
cat errcode "0"
SQL SELECT * from TABLEA
if (errcode == "0")
        msg emsg "Hit F2 for more rows."
else if errcode == "000100:000000"
        msg emsg "Done"
else
        msg emsg "Error"
```

DBMS START and Error Signalling

An argument greater than 1 in a DBMS START statement causes one fewer than that number of rows of selected data to be ignored in a query. Any errors that internal fetches of those rows may cause are also ignored. This means that if a DBMS START command causes a particular query to ignore all its selected rows, then the execution of that query will not cause any SQL errors to be signalled. End-of-SELECT messages are still available and can be trapped into a JAM variable as usual.

III. Text Datatype and Word-Wrapped Arrays

JAM variables have a length limit of 255 characters. Word-wrapped JAM array fields are used to handle data longer than that length. This is useful for handling TEXT database data types. JAM arrays that are not fields in the current form will not be treated having word-wrapped edit.

SELECTing into Word-Wrapped Arrays

If a word-wrapped array is one of the destinations for a SELECTed column, fetches are done one row at a time, with the word-wrap edit invoked on the relevant fields.

Updating from a Word-Wrapped Array

There are 2 ways to get the value of a full JAM array (i.e., all the elements) as 1 string into a JAM/DB*i* SQL statement. Let JA be an array:

```
SQL Insert into TABLE1 (LONGCOL) values (":JA")

SQL Insert into TABLE1 (LONGCOL) values (::JA)
```

The first example uses colon expansion to get the value of the array. In the second case, JA has to be a word-wrapped array.

Note that in the first example, the INSERT command is restricted in size by the JPL statement length limitations (approx. 2,000 characters). Therefore, when storing 'Long' values into a table, the second method is recommended.

IV. Customizing Query Result Destinations

The following commands specify where to send the results of a database query:

DBMS REDIRECT *cursor_name* TO *file_name* [TEE]

DBMS REDIRECT *cursor_name*

DBMS CATQUERY *cursor_name* TO *jam_fvar*

DBMS CATQUERY

DBMS OCCUR [*number_1* / *current*] [MAX *number_2*]

DBMS OCCUR

where:

cursor_name	is the name of a previously declared cursor (see Chapter I)
file_name	is the name of a file, including full path if not in current directory. There can be no embedded spaces in the file name.
jam_fvar	is the name of a JAM field or variable that will be active during execution of the intended query.
number_1 *number_2*	Integers greater than 0.

These commands allow a query's results to be sent to a file or a JAM variable, bypassing any column mapping. They also allow the user to specify a start index into the destination array and maximum number of rows to fetch.

In addition, query result column mapping now supports JPL variables as well as the other kinds of JAM variables.

Note: The term "fetch-execution" will be used to denote the execution of any of the following JAM/DB*i* statements.

SQL SELECT...
DBMS EXECUTE...
DBMS CONTINUE...

DBMS Redirect

JAM/DB*i* 4.7 includes a rudimentary report mechanism that allows the results of a query to be sent to a file. The command to specify this is associated with a cursor:

DBMS REDIRECT *cursor_name* TO *file_name* [TEE]

The optional TEE indicates that the query results should also go to its specified or default JAM variables.

If the query is going only to a file ("file-only mode"), or when the CATQUERY command is in effect (see below), the column widths used are derived from the table width definitions. If the results are also going to JAM fields (TEE), then the JAM widths are used. Columns in the file are separated by 2 spaces.

When the TEE option is present each fetch-execution of the query (using DBMS EXECUTE or DBMS CONTINUE) will send only the result rows fetched into the JAM variables or fields into the file. Several DBMS CONTINUEs may be needed to complete the report. In the file-only mode, just executing the cursor (DBMS EXECUTE) will send all the result rows into the named file.

The file *file_name* is opened when the REDIRECT command is executed, and all subsequent executions of the cursor add to the file. Any file of the same name that existed before the REDIRECT command is executed is over-written. The named file remains open, and associated with the specified cursor, until the

cursor is either closed or redeclared, or the cursor is redirected to another file, or the following command is issued:

DBMS REDIRECT *cursor_name*

A number of files may be open at the same time, associated with different cursors, subject to machine limits. Redirects of a cursor not associated with a SELECT statement produce no results.

DBMS Catquery

Normally, the result columns of a SELECT statement get mapped to corresponding JAM variables or fields of the same name. The following command allows a full query result row to be fetched into a single JAM field or variable, by passing any default or specified individual column mappings:

DBMS CATQUERY *cursor_name* TO *jam_fvar*

After this command is executed all subsequent executions of a SELECT statement (DBMS EXECUTE, DBMS CONTINUE, SQL SELECT) will send their results to *jam_fvar*. This mode will remain in effect until the following command is executed:

DBMS CATQUERY *cursor_name*

A single fetch-execution will attempt to fill the destination array field or variable by returning as many rows as the array dimension. A DBMS CONTINUE will fetch the next batch. If the destination field is word-wrapped, only 1 row will be fetched per fetch-execution. Individual columns in the query result are separated by 2 spaces.

Note that *jam_fvar* must be accessible from the place that the DBMS CATQUERY is issued for it to work, otherwise a warning message is flashed on the screen and catquery mode for that cursor is reset. It is therefore advisable to put the CATQUERY destination into the data dictionary.

DBMS Occur

When the JAM destination for a query is an array or set of parallel arrays, the DBMS OCCUR command may be used to specify a part of the array to be used as the query destination. The default start index (the destination for the first fetched row of a fetch-execution) is 1. The default maximum number of rows to fetch in a particular fetch-execution equals the number of complete rows that the destination can hold (see your JAM/DB*i* manual). These 2 values can be modified by the following command:

DBMS OCCUR [*number_1* / *CURRENT*] [MAX *number_2*]

The first parameter in the OCCUR command is the start index. This may be a number (*number_1*), which will be the new start index, or CURRENT, in which case the start index will be whatever row the cursor is on at the time of the fetch-execution. The second parameter (MAX *number_2*) specifies the maximum number of rows to fetch. Both parameters are optional. However, if both are present, MAX *must* come second.

The new values of start index and MAX come into effect as soon as the DBMS OCCUR statement is executed. These values affect any subsequent fetch executions (including CONTINUE). To reset to default mode, use:

DBMS OCCUR

V. Miscellaneous

Suppressing Repeating Values in a Query

The following DBMS command will cause JAM/DB*i* to suppress repeating values in columns of a query result:

DBMS SUPREPS *int {, int}**

The integer arguments represent the column numbers, in any order, by position in a SELECT. The first column number is 1. When a * is used in a SELECT statement (e.g., SELECT * FROM...) the column numbering is according to the SELECT statements output. This order usually follows that in the table definition. Column suppression is an ON/OFF command, and takes effect from the next fetch-execution (including cursors and CONTINUEs). It may be combined with any of the query destination customizing commands of the previous chapter.

```
DBMS SUPREPS 1, 3

SQL SELECT city, team, venue FROM homesites
DBMS CONTINUE
      ...

DBMS SUPREPS
```

In the above example, the columns for `city` and `venue` will have their repeating values suppressed, producing an output like the following (if the table is already sorted on `city` as the major key and `venue` as the secondary key):

N.Y.C.	Knicks	Garden
	Rangers	
	Globetrotters	
	Mets	Shea Stadium
Boston	Celtics	Garden

E. Ruthfd	Nets	Byrne Arena
	Devils	
	Giants	Giant Stadium
	Jets	

A SUPREPS command over-rides all previous commands. A DBMS SUPREPS without any arguments (as in the last line of the example) resets and turns suppression off.

Any column numbers greater than 0 may be given as arguments. Only those numbers relevant to a particular query will be considered. For example, `DBMS SUPREPS 7, 1, 3` would produce the same result as above. If a subsequently issued query had a seventh column, that also would have been suppressed. At most, 25 columns can be targeted by a SUPREPS command.

Printing a File

The following DBMS command may be used to print a file:

DBMS PRINT *file_name*

where

file_name is the name of a file, including the full path if not in the current directory. There can be no embedded spaces in the file name.

The JAM configuration variable SMLPRINT should be set to the appropriate print command string (see your **JAM Configuration and Utilities** guide).

JAM Variables and JPL Variables

A JAM variable denotes any of the 3 kinds of variables supported by JAM:

- Local JPL variables;
- JAM screen fields or arrays;
- JAM data dictionary variables.

JAM allows the same name to have 3 different definitions in these 3 locations. The above sequence also gives the order of precedence in which the corresponding definitions take effect, with JPL variable definitions superceding the rest.

JAM/DB*i* now fully supports variable substitution from JPL variables for all SQL statements.

NULL Values

When writing into the database (e.g., INSERT, UPDATE), an empty string will not be interpreted as NULL. In order to enter NULL values into a table, the word NULL must be specified. For instance,

```
SQL INSERT INTO PLAYOFF (TEAM, CITY, DATE) \
      VALUES (NULL, ':City', ':Date')
```

a NULL is being inserted into the character field TEAM. If the JAM fields `City` and `Date` are empty, those values will be translated as empty strings.

When a return value from a query is NULL, the string 'NULL' will be displayed in the corresponding JAM variable.

Scrolling Through the SELECTed Rows

Scrolling is allowed unless a buffer has been setup to store the rows which are already fetched. Use the following command to set a buffer before executing the query.

DBMS SET_BUF *number of rows to allow in buffer* [*cursor_name*]

Buffering takes up memory, so set a buffer only when CONTINUE_UP, CONTINUE_TOP, or CONTINUE_BOT is desired. Reset the buffer as soon as it is no longer necessary. Buffering will be reset by passing integer value zero as argument to DBMS SET_BUF.

The following commands scroll through the selected set of rows:

DBMS CONTINUE [*cursor_name*]

DBMS CONTINUE_DOWN [*cursor_name*]

DBMS CONTINUE_UP [*cursor_name*]

DBMS CONTINUE_TOP [*cursor_name*]

DBMS CONTINUE_BOT [*cursor_name*]

The first two are equivalent, and try to fetch the next page of rows into JAM's current view window (e.g., an on-screen array field). See description of SELECT for determining number of rows comprising a view window.

CONTINUE_UP will try to fetch the previous page of rows.

CONTINUE_TOP displays the first page of rows.

CONTINUE_BOT displays the last page of rows.

The following example shows the sequence of command to be issued:

```
DBMS SET_BUF 100

SQL SELECT city, team, venue FROM homesites
DBMS CONTINUE_UP
      ...

DBMS SET_BUF 0
```

Normally, there is no overlap between the rows displayed (i.e., fetched into the **JAM** destination) before and after the CONTINUE_UP (or CONTINUE_DOWN) command is issued. However, when that command positions **JAM**'s current view window at the top (bottom) of the select set, the first (last) row of the set is fetched into the first (last) element of the destination; the rest of the rows fill up the rest of the destination. When the number of rows in the select set is less than the size of the view window, the elements are filled from top down.

If the buffer set up is not large enough to hold all return rows, the top of the buffer will be cleared as newly fetched rows are stored. In that case, CONTINUE_TOP will not return the first selected rows, but the first row in the row buffer. For instance, if the row buffer is of size 100, and 120 rows are fetched, then CONTINUE_TOP will starting fetching from row 21 instead of the first row (which has been cleared).

The DBMS COUNT... command can be used to determine the number of rows fetched after each CONTINUE_*.

If the column repeat suppression mode is on (see DBMS SUPREPS), scrolling up or to the top or bottom will always display all the columns of the row in the first element, suppression always being executed reading from the top down.

Selecting BINARY Datatypes

Binary data cannot be retrieved into JAM variables or fields. The JAM/DB*i* library (`sybdbi.a`) contains a global variable named `DBi_image`, of type DBBINARY*, which will point to the retrieved data (DBBINARY is a Sybase DB-Library datatype). This variable may be freed after being processed; if not, it will not be freed until the next binary column selection is invoked. Only 1 row will be retrieved if the select statement involves a column of binary type. No matter how many binary fields are selected, only the first binary column will be retrieved.

Browse Mode

(Browse Mode is not available for servers running SYBASE Ver 3.X)

JAM/DB*i* supports the BROWSE MODE of SYBASE DB-Library by supplying the following commands:

DBMS BROWSE *select_statement*

DBMS UPDATE *cursor_name* SET *col1 = exp1 [, col2 = exp2...]*

These allow browsing selected rows and updating their values one row at a time. For example,

```
DBMS COUNT row
DBMS  BROWSE  select  field1  =  PRODUCT.PRICE  from\
      CLIENT, POSITIONS where PRODUCT.PRICE\
      = POSITIONS.PRICE.
while (row > 0)
{
      DBMS UPDATE PRODUCT set PRICE = :field1 + 2
      DBMS CONTINUE
}
```

Handling Stored Procedures and Their Results

```
DBMS PROC [cursor_name] EXEC\
      <batch command for stored procedure>

DBMS CONTINUE [cursor_name]

DBMS NEXT [cursor_name]

DBMS CANCEL [cursor_name]

DBMS FLUSH [cursor_name]

DBMS RETURN var
```

(DBMS RETURN is available only for servers running SYBASE Ver 4.X)

There may be multiple SQL statements in a stored procedure. All SQL statements will work the same way as normal JPL SQL statements. Data returned via select statements will be mapped to JAM variables. If the number of rows of fetched data is beyond the size of the destination, the stored procedure will be suspended until you call either,

`DBMS CONTINUE`	to continue fetching data, or
`DBMS NEXT`	to flush the pending query and start executing the next command, or
`DBMS CANCEL`	to cancel the stored procedure:

If, however, there is no need to call DBMS CONTINUE, JAM/DB*i* will immediately continue to execute the next SQL statement in the stored procedure without waiting for a DBMS NEXT. A pending stored procedure will automatically be cancelled by any DBMS or SQL statement (other than DBMS CONTINUE, DBMS FLUSH, or DBMS NEXT) that uses the same cursor.

There are several ways to execute a stored procedure:

In the following example, the stored procedure `proc1` will be executed using the cursor `cursor_a`:

```
DBMS PROC cursor_a EXEC proc1
```

In the following example, stored procedure `proc2` will be executed using default cursor and the return status of the `proc2` will be trapped in the JAM variable `status`:

```
DBMS RETURN status
DBMS PROC EXEC proc2
```

(DBMS RETURN is only available for servers running SYBASE Ver 4.X)

Return-value parameters are supported only by SYBASE Ver 4.X servers. In the following example, the stored procedure `proc3` will receive parameters 1, 2, and 50; the return parameter of the stored procedure will be trapped in the JAM variable `result`:

```
DBMS PROC EXEC declare @const int\
select @const = 50\
exec proc3 1, 2, @result = @const output
```

`DBMS NEXT [cursor_name]`

> *cursor_name* is the identifier of an open cursor. If it is null, the default cursor will be used.
>
> This command flushes the select statement pending in the stored procedure, and then executes the next SQL statement in the stored procedure.

`DBMS CANCEL [cursor_name]`

cursor_name is the identifier of an open cursor. If it is null, the default cursor will be used.

This command cancels a suspended stored procedure.

`DBMS FLUSH [cursor_name]`

cursor_name is the identifier of an open cursor. If it is null, the default cursor will be used.

This command flushes any rows pending in the stored procedure or SQL SELECT statement associated with the given cursor. It is typically used when the last statement in a stored procedure is SELECT.

NOTE: No return values or return parameters are set until the stored procedure terminates. In particular, if any selected rows are pending, the return value and parameters will be unavailable. If the number of rows fetched by SELECT or DBMS CONTINUE exactly matches the number of elements in the destination variables, the SELECT is considered to be pending, since the "no more rows" condition cannot be detected until the next fetch. Use DBMS FLUSH (or DBMS NEXT, or DBMS CONTINUE) to ensure that no SELECT is pending.

Default Cursors

In JAM/DB*i* for SYBASE, a user is automatically logged on to the server with two cursors (DBPROCESSes) so that a query (a JPL SQL select statement) can be interrupted by other non-select JPL SQL statements. There is an option for the DBMS LOGON command, -C<num>, which changes this default feature. When <num> is 1, only one default cursor will be used when logging on. Using -C<num> with any number other than 1 will assume two cursors; calling DBMS LOGON without the -C option will do the same.

If a user is logged on with only *one* cursor, he may *not* use certain JAM/DB*i* features:

- DBMS BROWSE and DBMS UPDATE command will be disabled.
- Whenever a query is interrupted, it will not be resumed.

However, the above features can be implanted with two named cursors.

Appendix B: Database-Specific Commands for SYBASE / SQL Server

Note: This appendix summarizes the database specific commands for the BETA release of JAM/DB*i* version 4.7 for SYBASE / SQL Server.

DBMS BEGIN [*transaction_name* [*cursor_name*]]

transaction_name is the name assigned to a transaction.

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

This command marks the starting point of a transaction, and must be followed by a DBMS COMMIT or DBMS ROLLBACK.

DBMS BROWSE *query*
(DBMS BROWSE is not available if server is running SYBASE Ver 3.X)

query is a select statement. See the Sybase command reference for the correct syntax for select statements.

This command uses the default cursor to perform a browse mode select. The browse mode select is similar to an ordinary select except that JAM/DB*i* only fetches 1 row at a time. The DBMS UPDATE command may be used to update the current row. DBMS CONTINUE will fetch the next row.

DBMS CATQUERY *cursor_name* [TO *jam_var*]

cursor_name is the identifier of an open cursor.

jam_var is the name of a **JAM** variable.

This command redirects the results of a query to a **JAM** variable, bypassing the normal **JAM/DB*i*** column mapping. The redirection remains in effect until you close the cursor, redeclare it, or execute DBMS CATQUERY *cursor_name* without the TO clause.

DBMS CLOSE CURSOR *cursor_name*

cursor_name is the identifier of an open cursor.

This command closes the specified cursor.

DBMS COMMIT [*transaction_name* [*cursor_name*]]

transaction_name is the name assigned to a transaction.

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

This command marks the end point of a transaction. It must be used following a DBMS BEGIN.

DBMS CONNECT *cursor_name* TO DATABASE *database_name*

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

database_name is the name of the database which the cursor will log onto.

This command create a login (with the login attributes used in DBMS LOGON) to the server using the specified database.

DBMS CONTINUE [*cursor_name*]

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

This command fetches the next page of rows.

DBMS CONTINUE_BOT [*cursor_name*]

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

This command fetches the last page of rows.

DBMS CONTINUE_DOWN [*cursor_name*]

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

This command fetches the next page of rows.

DBMS CONTINUE_TOP [*cursor_name*]

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

This command fetches the first page of rows.

DBMS CONTINUE_UP [*cursor_name*]

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

This command fetches the previous page of rows.

DBMS COUNT *count_var*

count_var is a **JAM** variable.

This command sets *count_var* to record the number of rows fetched.

DBMS CREATE_PROC [*cursor_name*] EXEC *<command to create stored procedure>*

This command creates a stored procedure.

DBMS CREATE_TRIGGER [*cursor_name*] EXEC *<command to create trigger for stored procedure>*

This command creates the trigger for a stored procedure.

DBMS DECLARE *cursor_name* CURSOR FOR *sql_stmt*

cursor_name is an identifier for a cursor.

sql_stmt is a Sybase SQL statement.

This command stores *sql_stmt* before actual execution.

DBMS DROP_PROC [*cursor_name*] EXEC *procedure_name1* [, *procedure_name 2*,...]

This command removes a stored procedure.

DBMS DROP_TRIGGER *trigger_name1* [, *trigger_name2*,...]

This command removes the trigger for a stored procedure.

DBMS ERROR [*code_var* [*message_var*]]

code_var and *message_var* are **JAM** variables. They may take any of the following forms:

id
id[int]
id[id]

where *id* is a **JAM** identifier and *int* is an integer. *id [int]* and *id[id]* are references to array elements.

This command causes JAM/DB*i* to store error codes and messages in the specified variables or array elements. Error trapping remains in effect until you execute the command DBMS ERROR with no arguments.

JAM/DB*i* will assign *code_var* a code which consists of two 6-digit fields separated by a colon (i.e., xxxxxx:yyyyyy). The first field is either the Database Library error code or the DataServer error code. When error trapping is on, all error and informational (severity=0) messages are trapped into *code_var* and *message_var*.

DBMS ERROR_CONTINUE

This command prevents JAM/DB*i* from aborting a JPL procedure when an error is detected.

DBMS EXECUTE *cursor_name*

cursor_name is the identifier of an open cursor.

This command executes the SQL statement specified in the corresponding DBMS DECLARE command.

DBMS LOGOFF

This command exits from Sybase server.

DBMS LOGON *[-t timeout] [-H hostname] [-U username] [-P password] [-I interface] [-S server] [-C 1 or 2]*

-t timeout Number of seconds that DB-Library waits for a login response before timing out. A timeout value of 0 represents an infinite timeout period. The default is 60 seconds.

-H hostname Allows the user to specify a host name, changing the value in the dynamic system table `sysprocesses`, if logging from a different

computer than usual. If no host name is specified, the current computer name is assumed.

-U username Allows the user to specify a login name. Logins are case sensitive.

-P password Allows the user to specify a password. Passwords are case sensitive.

-I interface Allows the user to specify the name and location of the interfaces file that is searched as part of the process of connecting to SQL Server. The named file contains the name and network address of every available SQL Server on the network. If this option is not used, `isql` looks for a file named `interfaces`.

-S server Allows the user to specify the name of the particular SQL Server with which to connect. This is the name that SQL Server looks up in the `interfaces` file.

-C [1 or 2] Allows the user to specify the number of default cursors. The default is 2.

Note: If a flag is given without a parameter value, the value is taken as NULL, or 0 for timeout.

This command connects the Sybase server.

DBMS OCCUR [*int1* | CURRENT] [MAX *int2*]

int1 and *int2* are integers greater than 0.

DBMS OCCUR *int1* specifies that *int1* is the first row of the array to be filled.

DBMS OCCUR CURRENT specifies that the array row where **JAM**'s cursor is located is the first row of the array to be filled.

DBMS OCCUR MAX *int2* specifies that *int2* is the maximum number of rows to be fetched.

DBMS PRINT *file_name*

file_name is the name of an existing file.

This command will print the contents of the file. You must have the SMLPRINT **JAM** configuration variable set to the appropriate command string (see **JAM Configuration and Utilities Guide**).

DBMS PROC [*cursor_name*] EXEC *cmd*

cursor_name is the identifier of an open cursor.

cmd is the batch command which executes a stored procedure.

This command executes a stored procedure and does column mapping to JAM if there is a query within the stored procedure.

DBMS REDIRECT *cursor_name* [TO *file_name* [TEE]]

cursor_name is the identifier of an open cursor.

file_name is the name of a file which DBMS REDIRECT will open.

This command redirects the results of a query to a file. Executing DBMS REDIRECT opens the file. If the file already exists, all previous data will be over-written. Subsequent executions of the cursor append query output to the file. The redirection remains in effect until you close the cursor, redeclare it, or execute the command DBMS REDIRECT *cursor_name* without the TO clause.

DBMS REDIRECT *cursor_name* TO *file_name* TEE directs the query results to both *file_name* and other specified or default **JAM** variables. If you do not use the TEE option, query results will go only to *file_name*.

DBMS RETURN *return_status_var*
(DBMS RETURN is available only if server is running SYBASE Ver 4.X)

return_status_var is a **JAM** variable.

This command set *return_status_var* as the destination for the return status of a stored procedure.

DBMS ROLLBACK [*transaction_name* / *savepoint_name* [*cursor_name*]]

transaction_name is the name assigned to a transaction.

savepoint_name is the name assigned to the savepoint of a transaction.

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

This command flushes all transactions since the last DBMS BEGIN or savepoint.

DBMS SAVE *savepoint_name* [*cursor_name*]

savepoint_name is the name assigned to the savepoint of a transaction.

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

This command sets a savepoint within a transaction.

DBMS SET_BUF *integer* [*cursor_name*]

integer is an integer greater than or equal to 0.

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

DBMS SET_BUF with a non-zero number will set the DBBUFFER option with the specified number of rows. This option must be set for DBMS CONTINUE_UP, DBMS CONTINUE_TOP, and DBMS CONTINUE_BOT commands to function.

DBMS SET_BUF with *integer* = 0 resets the buffer.

DBMS START *row_number*

row_number is a positive integer.

This command causes JAM/DB*i* to read and ignore a specified number of records (*row_number* - 1) before fetching the remaining query results into JAM.

DBMS SUPREPS *int1* [, *int2* ... *int25*]

int1, *int2*, ... *int25* are integer references to columns in a SELECT statement. Thus, DBMS SUPREPS 1, 3 refers to the first and third columns in a subsequent SELECT.

This command suppresses repeating values in the column when the data are fetched into a JAM array. Suppression of repeated values remains in effect until you execute a DBMS SUPREPS with no arguments.

DBMS UPDATE *table_name* SET *col_1* = *expression 1* [, *col_2* = *expression 2*...]

(DBMS UPDATE is not available if server is running SYBASE Ver 3.X)

This command is used to update a row in a browsable table under browse mode. For this command to be invoked:

- The table must be browsable (see DB-Library Reference Manual);
- You must have previously executed DBMS BROWSE

DBMS USE *database_name*

This command changes the current database.