New Features in JAM/DBi for ORACLE

Release 4.8

Contents

I. Multiple Cursors

In the following,

cursor_name is an identifier, consisting of non-blank characters, with maximum same as a **JAM** variable.

sql_statement is an SQL statement, possibly containing variables (syntax is database specific).

var is a **JAM** variable reference. It may have one of the following three forms:

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

where id is a **JAM** identifier, and int is an integer. The last 2 forms are array element references. Throughout this document, "**JAM** variable" will be used to denote any of the different kinds of variables supported in **JAM**, i.e., JPL variables, screen fields, and data dictionary variables. When the same name is defined in more than one place, JPL variables have the highest order of precedence, and data dictionary definitions the lowest.

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

DBMS DECLARE *cursor_name* CURSOR FOR *sql_statement*

DBMS EXECUTE *cursor_name* [USING *var* [, *var*]]

DBMS CONTINUE [*cursor_name*]

DBMS CLOSE CURSOR *cursor_name*

To use a cursor, the above sequence of statements must be followed. Any number of EXECUTEs and CONTINUEs may be issued, perhaps with different variable names. A CONTINUE will always try to continue the last EXECUTE for that cursor name. A CONTINUE without a cursor name tries to continue the last non-cursor SELECT statement.

The **JAM** variables in a DBMS EXECUTE are the arguments for the variables in that cursor's associated SQL statement. The SQL variables are replaced by the value of the corresponding **JAM** variables, as determined by the order, going from left to right. Thus the first **JAM** variable corresponds to the first SQL variable, and so on. If there are not enough **JAM** arguments, an error is generated. Any extra arguments are ignored.

When an index is supplied for a **JAM** variable used in a DBMS EXECUTE, the corresponding element of that **JAM** array is used. If the array is word-wrapped, the value used is the value of all the elements concatenated together, starting from the given index, to the end of the array. If no index is supplied and the variable is an array, then the default index of "1" is assumed.

Using Multiple Cursors

A cursor allows an SQL statement to be precompiled, before actual execution. At compile time (DBMS DECLARE...CURSOR...) the

statement may use variables instead of constants. The values for the variables are supplied at the time of execution. For example, in the statement:

```
DBMS DECLARE nba2 CURSOR FOR \
insert into TEAMS (TEAM, CITY) values (::TEAM, ::CITY)
```

JPL will replace the double-colon with a single colon, which is the required prefix for variables. The insert gets compiled with 2 arguments (:TEAM and :CITY) and associated with the cursor nba2. It can be executed by simply supplying the **JAM** variable names which contain the values for :TEAM and :CITY. For example:

```
DBMS EXECUTE nba2 USING Team, City
```

where Team and City may be 2 fields. The insert statement can be executed a number of times, using different arguments, without redeclaring it, and thus save the compilation time.

A cursor name is just an identifier. Any set of non-blank characters are allowed in the name. The name may be as long as a **JAM** variable name.

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)
```

Of course, a SELECT statement may also have variables in its WHERE clause, allowing slight modifications of the query with each DBMS EXECUTE. For example:

```
DBMS DECLARE nbateam CURSOR FOR \
SELECT * FROM NBATEAMS WHERE TEAM = ::teamname
DBMS EXECUTE nbateam USING NAME
DBMS EXECUTE nbateam USING TNAME
```

In the above example, "NAME" may be a JAM field, and TNAME a JPL variable.

Note that at the time of DECLAREing a SELECT cursor, JAM/DB*i* will map the columns of the target list into JAM fields or variables. Therefore, as far as the SELECT destination names are concerned, the contexts of the DECLARE and the EXECUTE should be equivalent. A safe way to do this would be to make the destinations data dictionary variables. Data dictionary items are available in every context, although their attributes may be over-ridden by local entities of the same name.

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.

II. Error Processing

There are 3 DBMS statements for handling errors and warnings:

DBMS ERROR_CONTINUE

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

DBMS WARN [*warn-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[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 still get 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, get displayed. However when error trapping is on, all errors and informational codes returned by the database get 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.

The behavior of DBMS WARN is described in the JAM/DB*i* 4.0 documentation.

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* get 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 getting 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.

Signalling End-of-SELECT

A database error code NO_MORE_ROWS (1403 in ORACLE) is signalled whenever an execution of a SELECT or a CONTINUE results in no new fetches. This error code will not be flashed at the bottom of the screen if default error processing is on. However, it may be trapped into a JAM variable (e.g., ERRCODE) using a statement like DBMS ERROR ERRCODE.

When a SELECT is being executed, the destination fields or variables are always cleared before performing any fetches. Thus an empty SELECT buffer will result in empty destination fields. On a CONTINUE however, if there is no data to be fetched, the destination fields or variables are not cleared.

DBMS START and Error Signalling

An argument greater than 1 in a DBMS START statement causes 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 cannot be word-wrapped.

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")

DBMS DECLARE tbl1 CURSOR FOR \
SQL Insert into TABLE1 (LONGCOL) values (::JAVAL)
DBMS EXECUTE tbl1 USING JA
```

The first example uses colon expansion to get the value of the array. The second sequence of commands uses a cursor and SQL variables. 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* 4.8 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 being 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 superseding 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 is interpreted as NULL. In the statement:

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

a NULL is being inserted into the integer field GAME_NBR. If the JAM fields City and Date are empty, those values will also be translated to NULL.

When executing a cursored statement, if any of the argument fields or variables are empty, they will be treated as NULLs.

Appendix B: Database-Specific Commands for ORACLE

NOTE: ORACLE converts column names and column alias names to upper case. When writing SQL SELECT statements, be sure that all JAM screen and data dictionary variables used as column destinations have UPPER CASE names, regardless of any column aliasing.

DBMS CATQUERY *cursor_name* [TO *jam_var*]

cursor_name is the identifier of an open cursor.

jam_var is the name of a JAM field or variable which will be active when the cursor is executed.

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

This command writes all transactions since the previous DBMS COMMIT (or DBMS LOGON) to the database.

DBMS CONTINUE [*cursor_name*]

cursor_name is the identifier of an open cursor.

This command fetches the next *n* rows of a query result into **JAM**, where *n* is the smallest number of array occurrences involved in the fetch.

DBMS CONTINUE with no argument continues the most recent query not associated with a cursor. DBMS CONTINUE *cursor_name* continues the query specified by *cursor_name*.

DBMS COUNT *count_var*

count_var is a **JAM** field or data dictionary variable.

This command returns the number of rows that were fetched into a **JAM** array and stores the result in *count_var*.

DBMS COUNT does not necessarily return the same value as ORACLE's COUNT function. The standard COUNT returns the number of values in a column which satisfy the SELECT conditions. DBMS COUNT, however, returns the number of rows fetched into **JAM**.

DBMS DECLARE *cursor_name* CURSOR FOR *sql_stmt*

cursor_name is an identifier for a cursor. It cannot contain blanks and can have the same maximum length as a JAM variable.

sql_stmt is an ORACLE SQL statement, which may contain variables.

This command precompiles *sql_stmt* before actual execution. The cursor remains open (i.e., the compiled *sql_stmt* remains available) until you execute a DBMS CLOSE CURSOR command.

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

code_var and *message_var* are JAM variables or field identifiers. 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.

DBMS ERROR_CONTINUE

This command prevents JAM/DB*i* from aborting a JPL procedure when an error is detected. DBMS ERROR_CONTINUE remains in effect until you execute a DBMS ERROR command.

DBMS EXECUTE *cursor_name* [USING *var1* [, *var2*...]]

cursor_name is the identifier for an open cursor.

var1, *var2* ... *varn* are JAM variable references. 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 executes the SQL statement specified in the corresponding DBMS DECLARE command. If the SQL statement contains variables, JAM/DB*i* substitutes the values of *var1*, *var2*, ... *varn*.

DBMS LOGOFF

This command closes an ORACLE session.

DBMS LOGON *user_id password* | *user_id/password*

This command opens an ORACLE session.

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

int1 and *int2* are integers greater than 0.

When the destination of a query is an array or set of parallel arrays, the default destination for the first row of a query result is the first row of the array. The maximum number of rows returned by a single fetch is the total number of rows in the array. DBMS OCCUR allows you to change these defaults.

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.

If you specify both a starting row and a maximum number of rows, MAX *int2* must be the second clause in the command. The destination parameters you specify remain in effect until you execute another DBMS OCCUR command. Executing DBMS OCCUR with no arguments restores the default destination parameters.

DBMS PRINT *file_name*

file_name is the name of an existing file. You must include the full path for files outside the current directory.

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 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. You must include the full path for files outside the current directory.

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 ROLLBACK

This command flushes all transactions since the last DBMS COMMIT or DBMS LOGON without writing them to the database.

DBMS START *row_number*

row_number is a positive integer.

This command causes JAM/DB*i* to read and ignore a specified number 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 WARN *warn_array*

warn_array is the identifier of an 8-element JAM data dictionary array.

This command causes JAM/DB*i* to place a "W" in each element of *warn_array* that matches a database warning. The ORACLE interface defines the elements as follows:

Element	Meaning
1	There was a warning issued
2	One or more fields was truncated on output
3	A null value was ignored in a function call
4	One or more output fields in a SELECT was not in the JAM data dictionary
5	UPDATE or DELETE did not have a WHERE clause
6	unused
7	Implicit rollback (e.g., a row was locked)
8	Row changed between SELECT and fetch of the row