# Chapter 7—SQL Programming

The SQL Browser lets you connect to your server database quickly and easily, and the SQL Form wizards let you build the interface to your server database. However you may want to modify the SQL forms created automatically or create forms from scratch to enhance your web and mobile apps. To do this, you use the *SQL methods*.

This chapter covers features of SQL programming that are common to all supported databases. The current version of Omnis Studio supports the following server databases:

1. Oracle 9i R2 and later
2. Sybase Adaptive Server Enterprise 12 and Sybase SQL Anywhere 9 and later
3. DB2 Universal Server /DB2 Express 9 and later
4. PostgreSQL 8 and later
5. MySQL 4.2 and later
6. SQLite data files v3 and later
7. all ODBC-compliant databases, such as MS SQL Server
8. plus you can access an Omnis database (data file) using Omnis SQL

For information about features that are particular to individual supported databases, see the *Server-Specific Programming* chapter.

(Note that support for JDBC has been removed in Studio 10, but the supporting files can be obtained by contacting support.)

## Overview

The *Object DAMs* (Data Access Modules) provide an object-oriented mechanism for establishing connections to a variety of SQL databases and enable you to perform multi-threaded database access as part of an Omnis Server in a web application. The DAM interface uses objects to represent a database session and session statements. These objects provide properties and methods that let you invoke the required database functionality. Using the object-oriented approach, an application creates object variables of a particular DAM class that are instantiated when the object is created by the component. This is known as a *session object* and represents a session of one of the supported DAM types. There is a group of common operations that apply to all session objects and a set of database specific operations based on the type of session object. For example, if an application requires an ODBC session, it uses an ODBCSESS session object. The session object controls the connection environment used to pass commands to the database server. The application creates a *statement object* in order to issue SQL. A statement object can be used for all types of statements, e.g., SQL, PL/SQL, cursors and remote procedures. There can be multiple statement objects which share a common context provided by a single session object.

In the Omnis Server multi-threaded environment there will be multiple clients each accessing session objects on the same server. Although it is possible to allow each client to use its own object this may use large amounts of system resource. Therefore, additional functionality is provided to create and maintain session object pools where each is of the same type. An application can allocate session objects from this pool. Each object in the pool is connected with the same attributes when the pool is created. This has an advantage in performance when a client requests an object since the connection is immediately available.

## Setting up a Database Connection

### Clientware

To connect to one of the supported databases, you may need to install the appropriate clientware on each machine that will be connecting to your server. The DAM uses this clientware to connect to the remote server.

We recommend which third party clientware you can use for each of the supported platforms and DAM connections. This information is available on the Omnis web site: **www.omnis.net/dams**. For each database type, a recommended driver name and version is provided. This is the driver which we have tested and certified against and which we recommend you to use.

For most platform/database combinations however, there will be other drivers which work comparably. In the event of technical support issues, it is the drivers listed on our web site which we ask faults be verified against.

### 64-bit DAMs

The DAMs provided with the 64-bit version of Omnis Studio use 64-bit architecture. This means that you will need to install separate 64-bit clientware where appropriate. The 64-bit DAMs are not interoperable with 32-bit client libraries and vice-versa. For single-tier and embedded DAMs, including DAMSQLITE, DAMOMSQL, DAMMYSQL, DAMPGSQL and DAMAZON, all necessary changes have been made. The 64-bit ODBC DAM requires the 64-bit ODBC Administrator library and should be used with 64-bit ODBC Drivers to ensure compatibility.

### INI files under macOS

Certain object DAMs available for macOS, namely DAMODBC, DAMSYBSE and DAMORA8, make use of “.ini” files in order to set system environment variables to be required by their associated client libraries. These files are named after the DAMs to which they apply and reside inside the Omnis package; in the Contents/MacOS/xcomp/ini folder.

If you are running Omnis from the command line (i.e. using the “open omnis.app” command), you can set environment variables from the context of the terminal window before starting Omnis, hence negating the need for these files. If used however, their values will override any existing values.

## Connecting to your Database

Aside from *hostname*, *username* and *password* information, the session templates in the SQL Browser contain all the necessary information required to connect to your server database automatically. However, to connect to your database programmatically you need to create a session object and log the session object onto the database.

The session object is the primary object that controls the session environment from which statement objects are created. This includes the connection, the transaction mode and any common statement properties such as the size of large object data chunks. The instance of an object is created either explicitly via the $new() external object method, or where it is first used. This initialises the object with default property values. When the object goes out of scope or is closed explicitly, depending on the session state, any statements are closed and the connection is logged off.

### Creating a Session Object

Create a variable of data type Object in the Variable Pane of the method editor and set the subtype to the session type required for the database connection.

|  |  |
| --- | --- |
| Connection | Session Type |
| Oracle | ORACLE8SESS |
| Sybase | SYBASESESS |
| DB2 | DB2SESS |
| PostgreSQL | PGSQLSESS |
| MySQL | MYSQLSESS |
| SQLite | SQLITESESS |
| ODBC | ODBCSESS |
| Omnis SQL | OMSQLSESS |

Note that most of these will only appear in the list of available session types if the relevant client software has been installed on your computer. DAM objects that fail to load due to missing or incompatible client software will leave an entry in the Trace Log when Omnis is started.

Any type of variable may be used to create a session object. However, a task variable is particularly suitable since it enables a variety of objects within your library to easily access the database session.

See the section on *External Objects* in the *Object Oriented Programming* chapter for further information about creating object variables.

### Logging on to a Session Object

The $logon() method is used to log on to the host server using a valid username and password. This establishes a connection that this session can then use to send commands to the server.

You can log on to a database using a method call of the form:

###### Do SessObj.$logon(pHostName,pUserName,pPassword,pSessionName) Returns #F

The parameters pHostName, pUserName, pPassword contain information required to connect to the host database. The values of these parameters vary according to the session object and database. If the pUserName and pPassword parameters are left empty, and depending on the session object type, the user may be prompted by the database client to enter the information. See the chapter on *Server-Specific Programming* for more information.

The optional parameter pSessionName is used to name the new session object and results in the session appearing under the SQL Browser tab and in the notational group; $sessions.

If the $logon() method is successful, it returns a value of kTrue, a connection to the database is created and the $state property is set to kSessionLoggedOn. The read-only properties $hostname, $username and $password of the session object are set to the values that were supplied.

For example, to log on to a session named “MySession” to a SQL Server data source named “MyDb” using an object variable “SessObj”

###### Do SessObj.$logon('MyDb','','','MySession') Returns #F

There is no limit to the number of session objects that you can create, except the limits imposed by memory resources and server restrictions.

#### Using Object References

There are two new properties $sessionobjref and $statementobjref of a list or row defined from a SQL table, and one new property of a session in $sessions called $sessionobjref. These are equivalent to $sessionobject and $statementobject, except that they work exclusively with object references.

### Logging Off from a Session Object

You need to log your session off from the server when you have finished with it using the $logoff() method.

The connection is dropped and the session $state property is set to kSessionLoggedOff. Depending on the session state, statements are cleared, cursors used by statements are closed and pending results are cleared. This call will fail if the session is not currently logged on. If it fails for any other reason, the connection may be in an undefined state and the current object should be destroyed. If there was an active transaction on this session, the behavior of the DBMS will determine if that transaction is committed or rolled back. The $hostname, $username and $password properties are cleared.

## Interacting with your Server

Once a user is logged into a server, they can make use of all the tables and views to which they have been granted access.

In order to send SQL commands to the server a statement object must first be created.

### Creating a Statement Object

A new statement object is created using the $newstatement() method. In order to return a new statement object in variable StatementObj with name “MySql” use

###### Do SessObj.$newstatement(‘MySql’) Returns StatementObj

The statement object can then be used to send commands and process results.

The variable StatementObj is defined as an object variable with no subtype. Again any type of variable may be used but a task variable is convenient to enable a variety of objects within your library to easily access the statement and its results.

If successful a statement object is returned otherwise this call has no effect possibly due to insufficient resources. The new object has a $statementname property value of “MySql” and defaults for all other properties.

If the parameter to $newstatement() is omitted, each statement name is automatically generated. These names are of the form “statement\_1”, “statement\_2”, etc.

### Mapping the Data

Before a client application can get any data from a server, it must set up a corresponding place in Omnis to hold the data. This involves mapping the structure of the data, including column names and data types. Typically, you do this using Omnis schema classes. You can define a schema to include all columns of the server table or view, or any subset of the columns. In addition you can create query classes that use columns from one or more schema classes.

You can use schema and query classes to define list and row variables to handle your server data. Information on creating schema, query, and table classes will be found earlier in this manual, as will details on using list and row variables.

#### Mapping Character Columns

The definition of Character columns in Omnis schema classes has changed in Studio 10 and now allows lengths from 0xffff to (100000000 - 1) to be stored correctly. In previous versions, the column sublen of 65535 or greater would have been mapped to 100000000.

### Sending SQL to the Server

To send SQL to the server, you can either write your own methods, or use the table instance methods that generate SQL automatically and handle both single row and bulk SQL transactions. SQL statements must first be prepared and then executed. If the statement returns results these may then be fetched.

#### Preparing a SQL Statement

When a SQL statement is prepared, it is sent to the server for verification and if valid is ready to be executed. A single line SQL statement may be prepared using the $prepare() method. Omnis sends to the server whatever you pass as a parameter to the method. It can be standard SQL or any other command statement the server can understand. For example

###### Do StatementObj.$prepare('SELECT \* FROM authors ORDER BY au\_lname,au\_fname') Returns #F

A value of kTrue is returned if the statement was successfully prepared, kFalse otherwise indicating that an error occurred. See the section on *Error Handling* for more information. Once a statement has been successfully prepared, the value of the $state property is set to kStatementStatePrepared.

A SQL statement may also be built up using a number of lines of method code as follows:

###### Begin statement Sta: SELECT \* FROM titles If iPrice>0   Sta: WHERE price>=[iPrice] End if Sta: ORDER BY title End statement Do StatementObj.$prepare() Returns #F

The *Begin statement* and *End statement* block contains the SQL statement each line of which is contained in an *Sta:* command. Other method commands such as *If*, *End if* etc. may be included inside the block in order to build up the SQL statement using logic.

Note that in this case a $prepare() method with no parameters is used to prepare the statement block.

When an Omnis web server is operating in a multi-threaded mode, the $prepare() method uses the statement buffer of the current method stack.

Once a SQL statement has been prepared it is possible to obtain the contents of the statement as a text string using the $sqltext property, for example

###### OK message The current SQL is {[StatementObj.$sqltext]}

The *Get statement* command returns a copy of the statement buffer created by the *Begin statement, End statement* and *Sta:* commands.  Get statement also replaces the bind variable place-holders in the copy of the statement it returns, with the normal Omnis syntax for bind variables ("@[...]").

#### Executing a SQL Statement

After a SQL statement has been prepared, using the $prepare() method, it can be executed. This is done using the $execute() method. For example

###### Do StatementObj.$execute() Returns #F

A value of kTrue is returned if the statement was successfully executed, kFalse otherwise indicating that an error occurred. If the statement $state property is kStatementStateClear prior to the execution of this method it will fail. Once a statement has been successfully executed, the value of the $state property is set to kStatementStateExecuted.

If the SQL statement generates results (e.g. a SELECT command), the $resultspending property of the statement is set to kTrue. These results may be retrieved using the $fetch() method.

Once a statement has been executed it may be re-executed as many times as is required using the $execute() method. There is no need to prepare it again unless a different SQL command is required. Re-executing a statement that has $resultspending set to kTrue will clear the results set however.

Alternatively, you can prepare and execute a SQL statement with a single command using the $execdirect() method. For example

###### Do StatementObj.$execdirect('SELECT \* FROM authors ORDER BY au\_lname,au\_fname') Returns #F

This method effectively performs a $prepare() followed by a $execute() method and if successful returns a value of kTrue. Once a statement has been successfully executed, the value of the $state property is set to kStatementStateExecDirect. It is not possible to re-execute a statement using $execute that has previously been executed using $execdirect().

### Fetching Results

When a SQL statement is executed that returns results (e.g. a SELECT statement), these can be retrieved using the $fetch() method in the form

###### Do StatementObj.$fetch(pTableRef,pRowCount,pAppend) Returns lFetchStatus

The results are placed in pTableRef, which may be a list or row variable. The pTableRef parameter may be omitted only if a previous $fetch() defined the row or list to use for returning data.

The pRowCount is a positive integer used to specify the number of results rows to return. It can also be set to kFetchAll to signal that all remaining rows in the result set should be returned. When pTableRef is a list, the number of rows returned is pRowCount or less. If pTableRef is a row variable, $fetch() always returns 1 row. If pRowCount is greater than the number of rows in the current result set, only the available rows are returned. If there are no more rows left in the current set, kFetchFinished is returned, otherwise kFetchOK is returned. If the pAppend parameter has a value of kTrue, the rows returned are appended to the list, or if kFalse, replace the previous contents of the list.

If the row or list is undefined, its column definition is created based upon the names of the columns in the result set and the results are retrieved. This can be forced by defining the variable as empty prior to fetching any rows. For example

###### Do iResultsList.$define()~ Do StatementObj.$fetch(iResultsList,9999) Returns lFetchStatus

If the variable was previously defined and the definition of the list or row columns do not match the data returned, any valid conversions are performed. If there are no more results pending, the $resultspending property of the statement is set to kFalse. An attempt to fetch when $resultspending is kFalse will result in a return status of kFetchError.

To limit the number of rows returned to a list use set the value of pRowCount to a value that will not make the list too big. For example

###### Do StatementObj.$fetch(iResultsList,100,kTrue) Returns lFetchStatus

The number of rows to be returned is in this case up to 100. If there were more than 100 rows in the results set, the extra rows will remain waiting to be fetched. The pAppend parameter is set to kTrue indicating that the results rows are to be added to the end of the list, preserving existing rows.

You can use the $linemax list property to limit the size of the list regardless of the number of rows in the results set. For example to limit the size of a list to 50 rows

###### Calculate iResultsList.$linemax as 50 Do StatementObj.$fetch(iResultsList,9999) Returns lFetchStatus

Any results generated by a statement will be available until either they have all been fetched or another query is executed using the same statement object, in which case the previous results set is destroyed.

To limit the result set based on memory rather than the list size, you can use the statement object’s $maxresultsetsize property which defaults to 100MB. $fetch() stops fetching when this limit is exceeded and lFetchStatus will be set to kFetchMemoryUsageExceeded.

#### Fetching Directly in to Omnis Variables

To return a row of data without using a list or row variable, the $fetchinto() method can be used in the form

###### Do StatementObj.$fetchinto(vVar1, vVar2,… vVarN) Returns lFetchStatus

$fetchinto() returns a row of data directly into the parameters supplied. A variable number of parameters can be supplied corresponding to each column in the result set .

#### Fetching in to an External File

The $fetchtofile() method lets you fetch a results set to an external file. It is the object DAM equivalent of the old-style command *Retrieve rows to file*, allowing one or more rows from a result set to be written directly into an external text file (export file).

$fetchtofile() implicitly opens and closes the export file, so also encompasses the old-style commands: *Set client import file name, Open client import file, Close client import file* and *Delete client import file*.

As with the $fetch() and $fetchinto() methods, a result set must exist before you call $fetchtofile(). The syntax used for $fetchtofile() is:

###### Do StatementObj.$fetchtofile(cFilename [,iRowCount=1][,bAppend=kTrue][,bColumnNames=kTrue][,iEncoding=kUniTypeUTF8/kUniTypeLatin1]) Returns Long integer

**cFilename** is the full path of the export file, formatted as required by the curremt operating system. The filename will usually be given the extension “.txt”

**iRowCount** is an optional parameter which specifies the number of rows to write to the export file. If iRowCount is less than the number of rows in the result set, $fetchtofile () can be executed again in which case the result set will be fetched from the last unread row. To retrieve all rows in the result set to the file, specify kFetchAll. If omitted, a single row is written.

**bAppend** is an optional parameter which specifies whether the existing file contents should be appended to or over-written. The default is to append the data (kTrue).

**bColumnNames** is an optional parameter which specifies that the first row of the file should contain the column names of the result set. This has no effect when bAppend is set to kTrue. The default behavior will automatically write column names to a file which is empty or being overwritten.

**iEncoding** is an optional parameter which specifies the type of encoding to be used. iEncoding should be one of the Unicode type constants and defaults to kUniTypeUTF8. The corresponding Unicode byte order marker (BOM) is written to the beginning of the file when the file is empty or when bAppend is set to kFalse.

$fetchtofile() automatically creates the export file if it does not exist or otherwise opens the file prior to writing the pending result rows into it. The file is closed immediately on writing the last row.

$fetchtofile() will return kFetchOk or kFetchFinished if the requested rows are successfully written to the export file, otherwise kFetchError is returned with the resulting error message being written to statement.$errorcode and statement.$errortext.

Two additional statement properties have been added which work in conjunction with the $fetchtofile() method. These are $columndelimiter and $rowdelimiterwhich are used to specify the characters that delimit columns and rows. It may be necessary to change these properties to match the format expected by an external application. $columndelimiter and $rowdelimiter accept multiple characters if required (15 characters maximum). The default value for $columndelimiter is kTab (chr(9)), and $rowdelimiter is kCr (chr(13)).

#### Batch Fetching

Where the DBMS is capable, it is possible to reduce network traffic when fetching multiple rows of data from the database by fetching several rows simultaneously (as a single *batch*), thus reducing the number of fetches required to return the entire result set.

There are three statement properties to manage batch fetching; $batchsize, $effectivebatchsize and $maxbuffersize.

By default $batchsize is set to 1, but this can be set to any long integer value. $batchsize should be set before the SQL SELECT statement is prepared. Changing the $batchsize automatically clears any results which may be pending on the statement object. When increasing the batch size, it should be noted that the memory resources required to receive the row set will increase proportionately.

$maxbuffersize can be used to limit the batch size such that for a given column of result data, the buffer size calculated as columnSize \* batchSize will not exceed the imposed maximum value. If the maximum buffer size would be exceeded, the batch size is revised downward at prepare time to accommodate the maximum number of result rows per fetch. The default value of $maxbuffersize is 32KB but will accept any long integer value greater than 255.

The resulting batch size is assigned to the $effectivebatchsize property (which is read-only).  
The value assigned to $batchsize is stored so that when the statement object is re-used, $effectivebatchsize can be calculated again.

It is not possible to return non-scalar data types using batch fetching. Such types include Oracle CLOBs and BLOBs which are not stored in the database directly but referenced by pointers (LOB locators). Thus, the raw data for each column value has to be fetched using a separate network transaction- negating the potential benefit of using batch fetching.

Batch fetching of large binary data is also prohibitively expensive in terms of memory requirements; only one or two rows could be fetched in most cases.

For similar reasons, batch fetching is also not supported for column data which is retrievable via chunks (chunking). If the database supports chunking of fetched data, the session property $lobthreshold should be set high enough to prevent possible chunking of any of the result columns.

When preparing a select statement which includes one or more non-scalar or chunk-able columns, $batchsize will be set back to 1 automatically. For example

###### Do mylist.$define() Do tStatement.$batchsize.$assign(3000) Returns #F ## Not implemented by all servers Do tStatement.$prepare('select \* from mytable') Returns #F OK message {$batchsize will be limited to [tStatement.$effectivebatchsize] for this result set} Do tStatement.$execute() Returns #F Do tStatement.$fetch(mylist,kFetchAll) Returns lStatus

The following object DAMs currently support batch fetching: DAMODBC, DAMORA8, DAMSYBSE & DAMDB2. For database servers which do not support batch fetching, the $batchsize and $maxbuffersize properties are also read-only.

#### Measuring Data Transfer

You can measure the amount of data (in bytes) that is received and sent through a session object since logon using the session properties $bytesreceived and $bytessent. The values can be reset by assigning zero to them. These properties apply to all DAMs.

### Describing Results

After executing a SQL statement, you can use the $results() method to obtain information about the columns contained in the current results set.

The only parameter required is a list variable where the information is to be placed. For example

###### Do StatementObj.$results(iResultsList) Returns #F

The returned list will be defined with the following columns and will contain one *row* for each *column* of the pending results set.

|  |  |  |
| --- | --- | --- |
| Col | Name | Meaning |
| 1 | ColumnName | Name of column. |
| 2 | OmnisData typeText | Omnis data type (description) |
| 3 | SQLData type | Equivalent standard SQL data type (CHARACTER, NUMBER, DATETIME, ...) |
| 4 | Length | Column width (for character columns) |
| 5 | Scale | Number of decimal places (for numeric cols), empty for floating numbers |
| 6 | Null | Nulls allowed (kTrue or kFalse) |

### Substituting Variables Into SQL Commands

You can substitute variables into the SQL statement using concatenation. You must supply quoted literals for character columns according to the rules imposed by your SQL server. For example

###### Do StatementObj.$prepare(con(“SELECT \* FROM authors WHERE state='”,iState,”' ORDER BY au\_lname,au\_fname”)) Returns #F

Note the use of double-quotes around the statement string, since it is necessary to embed single quotes around the value for the character column.

If you are using the *Sta:* command to build up a statement, a variable may be substituted using square bracket notation. For example,

###### Begin statement   Sta: SELECT \* FROM authors

###### If iState<>''     Sta: WHERE state='[iState]'   End if   Sta: ORDER BY au\_lname,au\_fname

###### End statement Do StatementObj.$prepare() Returns #F

Note that using these types of substitution you can only substitute variables containing data that can be represented by characters (i.e. character and numeric variables.) For other types of data including binary, pictures and lists you must use bind variables.

### Bind Variables

A *bind variable* allows an Omnis variable to be passed to a SQL statement. Instead of expanding the expression, Omnis associates, or binds the memory address of the variable to a SQL variable. To specify a bind variable you place an @ before the opening square bracket. Omnis evaluates the expression and passes the value to the server directly rather than substituting it into the SQL statement as text. You can also use this syntax to bind large fields such as binary, pictures and lists into a SQL statement. For example

###### Do StatementObj.$prepare('SELECT \* FROM authors WHERE state = @[iState] ORDER BY au\_lname,au\_fname') Returns #F Do StatementObj.$prepare(‘INSERT INTO authors (au\_lname,au\_fname) VALUES (@[iRow.au\_lname],@[iRow.au\_fname]’) Returns #F

Alternatively using the *Sta:* command:

###### Begin statement   Sta: SELECT \* FROM authors   If iState<>''    Sta: WHERE state=@[iState]   End if   Sta: ORDER BY au\_lname,au\_fname End statement Do StatementObj.$prepare() Returns #F

Note that you do not need to place quotes around a value to be substituted by a bind variable. You must include quotes when using square bracket notation to substitute character variables, but you don’t need to when using bind variables.

Not all database systems allow bind variables; in these cases Omnis will behave as though they do, but will instead perform literal expansion as though you had entered square bracket notation instead of bind variables

Generally, using bind variables results in better performance than using square bracket notation and is more flexible with respect to data representation. You should use square bracket notation only when the notation expression evaluates to a part of a SQL statement other than a value reference- such as an entire WHERE clause, or where you know that simple value substitution is all you need.

If you are inserting NULL data into the database, you should use bind variables, to ensure that SQL nulls, rather than empty strings are inserted.

Be careful to ensure when using a variable such as a local or instance variable, that it will still exist at the time when the SQL is executed using the $execute() method. This is not usually a problem when the $execute() method follows immediately after the $prepare, but may be if the $execute() method is located elsewhere (i.e. in another method.) It does not matter if an instance variable is not in scope in the method where the $execute() is located so long as the variable has not been destroyed by closing the instance that owns it.

Once a statement containing bind variables has been prepared, it can be repeatedly executed using the $execute() method. The values of the bound variables may be changed before each call to $execute(), allowing different data to be passed each time. This can greatly speed up the process of, say, inserting many rows into a server table within a loop.

### Constructing SQL Queries From Row Variables

Several methods are provided that allow you to construct SQL queries based on the definition of the columns of a row or list variable. These methods return partially complete SQL text that varies according to the session object in use and always includes all columns from the row or list variable.

#### Pre-V30 SQL Functions

Existing users should note: The Pre-V30 SQL functions insertnames(), selectnames(), updatenames(), and wherenames() have been removed from the Catalog (F9/Cmnd-9) in Omnis Studio 6.x onwards, but they can still be used in your code. In addition, the createnames() and server() functions have been removed from Omnis, including the Catalog, and will no longer work in code in Omnis Studio 6.x onwards.

#### $createnames()

To create a new table on the server, you can use the $createnames() method. $createnames() returns a text string which is a comma delimited list of column names and data types. Use is of the form:

###### Calculate lColList as SessObj.$createnames(pRowRef)

Parameter pRowRef is a row or list variable.

This can be used in a SQL CREATE TABLE statement, for example to create a new database table called “publishers” based on the definition of the columns in row variable “iPublishersRow”

###### Do StatementObj.$execdirect(con('CREATE TABLE publishers (', SessObj.$createnames(iPublishersRow),')')) Returns #F

Note that you need to include the parentheses around the column list because they are not provided by the method.

Depending on the session object, this will create a SQL command similar to

###### CREATE TABLE publishers (pub\_id varchar(4),pub\_name varchar(40),city varchar(20),state varchar(2),country varchar(30))

#### $insertnames()

To insert a row into a table, you can use the $insertnames() method. $insertnames() returns a text string that is a comma delimited list of column names and values to be inserted. Use is of the form:

###### Calculate lColList as SessObj.$insertnames(pRowRef)

Parameter pRowRef is a row or list variable. If it is a list variable, the values from the current line (pRowRef.$line) are used.

This can be used in a SQL INSERT statement. For example, to insert a row into the table “publishers” containing data from the row variable “iPublishersRow”:

###### Do StatementObj.$execdirect(con('INSERT INTO publishers ', SessObj.$insertnames(iPublishersRow))) Returns #F

Note that the method provides the parentheses around the column and values lists and the “VALUES” clause automatically.

This will create a SQL command similar to

###### INSERT INTO publishers (pub\_id,pub\_name,city,state,country) VALUES (@[iPublishersRow.pub\_id], @[iPublishersRow.pub\_name], @[iPublishersRow.city], @[iPublishersRow.state], @[iPublishersRow.country])

#### $updatenames()

To update a row in a table, you can use the $updatenames() method. $updatenames() returns a text string that is a comma delimited list of column names and values to be updated. Use is of the form:

###### Calculate lColList as SessObj.$updatenames(pRowRef)

Parameter pRowRef is a row or list variable. If it is a list variable, the values from the current line are used.

This can be used in a SQL UPDATE statement. For example, to update a row in the table “publishers” containing data from the row variable “iPublishersRow”:

###### Do StatementObj.$execdirect(con('UPDATE publishers ', SessObj.$updatenames(iPublishersRow),' WHERE pub\_id = @[iPublishersRow.pub\_id]')) Returns #F

Note that the method provides the “SET” clause automatically.

This will create a SQL command similar to

###### UPDATE publishers SET pub\_id=@[iPublishersRow.pub\_id], pub\_name=@[iPublishersRow.pub\_name], city=@[iPublishersRow.city], state=@[iPublishersRow.state], country=@[iPublishersRow.country] WHERE pub\_id = @[iPublishersRow.pub\_id]

#### $selectnames()

To select all columns in a table, you can use the $selectnames() method. $selectnames() returns a text string that is a comma delimited list of column names to be selected. Use is of the form:

###### Calculate lColList as SessObj.$selectnames(pRowRef,pTableName)

Parameter pRowRef is a row or list variable and pTableName is an optional parameter (default empty) that will prefix column names with the specified table name.

This can be used in a SQL SELECT statement. For example, to select all columns in the table “publishers” using the column definition of the list variable “iPublishersList”:

###### Do StatementObj.$execdirect(con('SELECT ', SessObj.$selectnames(iPublishersList),' FROM publishers ORDER BY pub\_name')) Returns #F

This will create a SQL command similar to

###### SELECT pub\_id,pub\_name,city,state,country FROM publishers ORDER BY pub\_name

#### $wherenames()

To locate a row or rows in a table to be updated or deleted you can use the $wherenames() method. $wherenames() returns a text string that is a comma delimited list of column names and values. Use is of the form:

###### Calculate lColList as SessObj.$wherenames(pRowRef,pTableName,pComparison,pOperator)

Parameter pRowRef is a row or list variable. If it is a list variable, the values from the current line are used. The remaining parameters are optional, pTableName will prefix column names with the specified table name, pComparison (default “=”) is used to specify an alternative comparison operator (e.g. >, <, >= etc.), pOperator (default “AND”) is used to specify an alternative logical operator (i.e. OR.)

This can be used in a SQL UPDATE or DELETE statement. For example, to delete a row in the table “publishers” where the column values exactly match all columns in the row variable “iPublishersRow”:

###### Do StatementObj.$execdirect(con('DELETE FROM publishers ', SessObj.$wherenames(iPublishersRow))) Returns #F

This will create a SQL command containing a WHERE clause similar to

###### DELETE FROM publishers WHERE pub\_id=@[iPublishersRow.pub\_id] AND pub\_name=@[iPublishersRow.pub\_name] AND city=@[iPublishersRow.city] AND state=@[iPublishersRow.state] AND country=@[iPublishersRow.country]

### Table and Column names

Table instances and session variables have a property called $quotedidentifier which determines whether or not table and column names are contained in quotes. If set to kTrue, table and column name identifiers returned from the $createnames(), $insertnames(), $updatenames(), $selectnames() and $wherenames() methods will be quoted “thus”, facilitating case-sensitive names and names containing spaces. The new property affects table instance methods as well as session object methods.

### SQL Errors

It is possible for an error to occur when a session or statement method is executed. The error may originate from the session object or in the database. The session or statement method will indicate that an error has occurred by returning the value kFalse. The value returned by these methods may be placed into any variable. The following example places the return value in the Omnis flag variable:

###### Do StatementObj.$execute() Returns #F

The methods will also return a generic (i.e. database independent) code and message describing the error in the properties $errorcode and $errortext. For example, in the event of an error occurring, SessObj.$logon() would return an error code in the property SessObj.$errorcode and an error message in SessObj.$errortext whilst StatementObj.$execute() would return an error code in StatementObj.$errorcode and message in StatementObj.$errortext. You can create a method to display the error code and message. For example:

###### OK message SQL Error {Code = [StatementObj.$errorcode], Msg = [StatementObj.$errortext]}

If smart lists are used, the history list row properties $errorcode and $errortext contain the value of StatementObj.$errorcode and StatementObj.$errortext for each row updated.

Additionally, you can obtain the native error code and message from the server using the session and statement properties $nativeerrorcode and $nativeerrortext. Some servers may return more than one combination of native error code and message for a single error. Currently only Sybase and Microsoft SQL Server behave in this way. When an error occurs the $errorcode and $errortext will contain a generic error from the session object and the $nativeerrorcode and $nativeerrortext will contain the first error reported by the server. If there are further error messages, $nativeerrorpending (a Boolean property of the statement object) will be set the kTrue. In order to access the next native error code and message, use the $nextnativeerror() method that will set the $nativeerrorcode and $nativeerrortext properties to the next error. You can repeatedly call the $nextnativeerror() method until all errors have been processed ($nativeerrorpending=kFalse). For example

###### While StatementObj.$nativeerrorpending Do StatementObj.$nextnativeerror() OK message SQL Error {Code = StatementObj.$nativeerrorcode], Msg = [StatementObj.$nativeerrortext]}]} End While

All server errors cause the statement method to return a value of kFalse. If you get an error from a method it does not prevent execution of further methods. You should always test the return value after execution of a session or statement method and take an appropriate action.

### Data Type Mapping

Omnis converts the data in an Omnis field or variable into the corresponding SQL data type. Since each DAM maps to a wide variety of data types on servers, each DAM determines the correct conversion for the current server. See the *Server-Specific Programming* chapter for details on how each DAM maps SQL data types to Omnis data types and vice versa.

#### DAMs support for 64-bit Integers

The Studio 6.0 DAMs perform additional data type mappings between Omnis 64-bit integers and the corresponding large integer data type on the database server. For most databases this will be BIGINT. The notable exception is Oracle which uses NUMBER(19,0) instead.

Note also that BIGINT UNSIGNED columns will be converted to signed 64-bit Integers when fetched into Omnis. In order to preserve such values, a *CAST(column as CHAR)* function can be used to fetch the value into a character field.

Where schemas and lists are defined using Integer 64-bit columns, the session object’s $coltext() and $createnames() methods now return the appropriate SQL data type. Integer 32-bit columns retain their previous behavior.

### Clearing Statements and Sessions

It is possible to clear individual statement objects back to their default state using the $clear() method. For example

###### Do StatementObj.$clear() Returns #F

When used with a statement, the $clear() method will cancel any pending results or operations on the statement object and set all properties to their default values (except for $statementname, $usecursor.) The statement object is placed in the kStatementStateClear state. If the statement is using a cursor, it will be closed.

When used with a session object the $clear() method will clear all statements controlled by the session object instance. For example:

###### Do SessObj.$clear() Returns #F

All of the statement objects are placed in a kStatementStateClear state. Depending on the session state this will clear all SQL text, close all cursors and destroy pending results of all statement objects owned by the session.

## Listing Database Objects

Omnis provides statement methods that enable you to access data dictionary information about any database to which you can connect using a session object. Using these methods, you can create database-independent code to list objects contained in your server database, no matter what its type. These methods work by creating results sets as though you had queried the information from the database. You then use the $fetch() method to read the results into Omnis.

### Listing Tables and Views

The $tables() method generates a results set containing details of tables available to the statement object. Use is of the form:

###### Do StatementObj.$tables(pType,pOwner) Returns #F

The parameter pType is used to indicate what types of object are to be listed and may contain kStatementServerTable to obtain details of all tables, kStatementServerView to obtain details of all views, or kStatementServerAll (default) to obtain details of all tables and all views. The parameter pOwner is used to list only objects belonging to a single named owner and defaults to all owners. For example to create a list of all available tables and views for all owners:

###### Do StatementObj.$tables() Returns #F Do StatementObj.$fetch(iResultsList,9999) Returns lFetchStatus

To create a list of all available views owned by DBO:

###### Do StatementObj.$tables(kStatementServerView,'DBO') Returns #F Do StatementObj.$fetch(iResultsList,9999) Returns lFetchStatus

The result set contains the following columns:

|  |  |  |
| --- | --- | --- |
| Col | Name | Description |
| 1 | Owner | Name of user that owns the database object |
| 2 | TableOrViewName | Name of table or view |
| 3 | TableType | Object type (kStatementServerTable or kStatementServerView) |
| 4 | Description | Remarks or description for the object where available |
| 5 | DamInfoRow | A row of database specific information about the table or view. This may be empty for some session objects |

### Listing Columns

The $columns() method generates a results set containing details of columns for a specified table or view.

The only parameter required is the name of the database table or view for which column details are required. For example to create a list of columns in the “authors” table:

###### Do StatementObj.$columns('authors') Returns #F Do StatementObj.$fetch(iResultsList,9999) Returns lFetchStatus

The results set contains the following columns:

|  |  |  |
| --- | --- | --- |
| Col | Name | Description |
| 1 | DatabaseOrCatalog | Name of database or catalog that contains the object |
| 2 | Owner | Name of user that owns the database object |
| 3 | ColumnName | Name of column |
| 4 | OmnisData typeText | Omnis data type (description) |
| 5 | OmnisData type | Omnis data type (notational) |
| 6 | OmnisDataSubType | Omnis data subtype (notational) |
| 7 | SQLData type | Equivalent standard SQL data type (CHARACTER, NUMBER, DATETIME, ...) |
| 8 | Length | Column width (for character columns only\*) |
| 9 | Scale | Number of decimal places (for numeric cols), empty for floating numbers |
| 10 | Null | Nulls allowed (kTrue or kFalse) |
| 11 | Index | Index exists for column (kTrue or kFalse) |
| 12 | PrimaryKey | Column is the primary key (kTrue or kFalse) |
| 13 | Description | Remarks or description for the column where available |
| 14 | DamInfoRow | A row of database specific information about the column. This may be empty for some session objects |

##### \*As of Studio 6.1, the default behavior for this column can be modified using an additional *flags* parameter. Refer to Statement Methods for details.

### Listing Indexes

The $indexes() method generates a results set containing details of index columns for a specified table. Use is of the form:

###### Do StatementObj.$indexes(pTableName,pType) Returns #F

The parameter pTableName is the table of which indexes are to be listed. The parameter pType is used to indicate what types of indexes are to be listed and may contain kStatementIndexUnique (default) to obtain details of unique indexes, kStatementIndexNonUnique to obtain details of non-unique indexes, or kStatementIndexAll to obtain details of all indexes. For example, to create a list of all indexes for the table “authors”:

###### Do StatementObj.$indexes('authors',kStatementIndexAll) Returns #F Do StatementObj.$fetch(iResultsList,9999) Returns lFetchStatus

The results set contains the following columns:

|  |  |  |
| --- | --- | --- |
| Col | Name | Description |
| 1 | DatabaseOrCatalog | Name of database or catalog that contains the object |
| 2 | Owner | Name of user that owns the database object |
| 3 | ColumnName | Name of column contained in index |
| 4 | IndexName | Name of index |
| 5 | Unique | Unique index (kTrue or kFalse) |
| 6 | ColumnPosition | Position of column (integer) in index (1 for normal index, 1,2,3… for column in compound index) |
| 7 | DamInfoRow | A row of database specific information about the index. This may be empty for some session objects |

### Building Schema Classes

Using the $makeschema() session method you can make a schema class automatically that matches the columns in a database table using a command of the form:

###### Do SessObj.$makeschema(pSchema,pTableName) Returns #F

The parameter pSchema is a reference to an existing schema class that will be overwritten with the definition from the server table pTableName.

For example, to create a schema called “scAuthors” from the server table “authors”:

###### Do $schemas.$add('scAuthors') Returns #F Do SessObj.$makeschema($schemas.scAuthors,'authors') Returns #F

Using the $tables() and $makeschema() methods you can obtain a list of tables on the server and build a schema class for each server table.

### Defining Lists from Server Tables

As of Studio 8.1.5, the session method; $definelistorrow() can be used to define a list or row variable directly from a named server table, i.e. without the requirement for a schema class. For example:

###### Do SessObj.$definelistorrow(iList1,’logon\_names’) Returns #F

Inside a table instance you can also pass $cinst as the list/row name. If the server table contains a primary key, $definelistorrow sets $excludefromwhere to kTrue for non-primary key columns.

## Remote Procedures

Omnis provides methods that enable you to list and execute remote procedures that exist in the database. Support for these methods varies from one database to another, see the chapter on *Server-Specific Programming* for more details.

### Listing Remote Procedures

The $rpcprocedures() method generates a results set containing details of remote procedures available to the statement object. Use is of the form:

###### Do StatementObj.$rpcprocedures(pOwner) Returns #F

The optional parameter pOwner is used to list only objects belonging to a single named owner and defaults to all owners. For example to create a list of all available remote procedures for all owners

###### Do StatementObj.$rpcprocedures() Returns #F Do StatementObj.$fetch(iResultsList,9999) Returns lFetchStatus

The results set contains the following columns:

|  |  |  |
| --- | --- | --- |
| Col | Name | Description |
| 1 | DatabaseOrCatalog | Name of database or catalog that contains the object |
| 2 | Owner | Name of user that owns the database object |
| 3 | ProcedureName | Name of remote procedure |
| 4 | DamInfoRow | A row of database specific information about the remote procedure. This may be empty for some session objects |

### Listing Remote Procedure Parameters

The $rpcparameters() method generates a results set containing details of all parameters required by a particular remote procedure. Use is of the form:

###### Do StatementObj.$rpcparameters(pProcedureName) Returns #F

The parameter pProcedureName is the name of the remote procedure. This parameter may take the form Database.Owner.Procedure name depending on whether database and owner qualifiers are supported. The Database and Owner are optional. The use of the qualifier is as follows.

|  |  |
| --- | --- |
|  |  |
| Only Procedure name specified | Return parameter information for all procedures with specified Procedure name in all available databases. |
| Owner.Procedure name specified | Return parameter information for all procedures with specified Procedure name owned by Owner in all available databases. |
| Database.Owner.Procedure name specified | Return parameter information for specified procedure owned by Owner in Database |

For example, to create a list of all parameters for remote procedure “byroyalty”:

###### Do StatementObj.$rpcparameters('byroyalty') Returns #F Do StatementObj.$fetch(iResultsList,9999) Returns lFetchStatus

The results set contains the following columns:

|  |  |  |
| --- | --- | --- |
| Col | Name | Description |
| 1 | OmnisData type | Omnis data type (notational) |
| 2 | OmnisDataSubType | Omnis data subtype (notational) |
| 3 | Length | Column width (for character columns) |
| 4 | PassType | How the parameter is used. One of the constants kParameterInput, kParameterOutput, kParameterInputOutput, or kParameterReturnValue. |
| 5 | C5 | Reserved for future use. |
| 6 | C6 | Reserved for future use. |
| 7 | DatabaseOrCatalog | Name of database or catalog that contains the object. |
| 8 | Owner | Name of user that owns the database object. |
| 9 | ParameterName | Name of the parameter. |
| 10 | OmnisData typeText | Omnis data type (description) |
| 11 | SQLData type | Equivalent standard SQL data type (CHARACTER, NUMBER, DATETIME, ...) |
| 12 | Scale | Number of decimal places (for numeric cols), empty for floating numbers |
| 13 | DamInfoRow | A row of database specific information about the parameter. This may be empty for some session objects |

##### Note that columns 1, 2 and 3 are the closest Omnis type to the server type specified in the server definition of the procedure parameter.

### Calling a Remote Procedure

Before you can call a remote procedure, it must first be registered with Omnis using the session method $rpcdefine().Use is of the form:

###### Do SessObj.$rpcdefine(pProcedureName,pList) Returns #F

The parameter pProcedureName is the case-sensitive name of the remote procedure that must exist on the server. If the procedure has previously been defined, the new definition replaces the old one. The parameter pList defines the parameters and the return value of the remote procedure. The list must have the same layout as that returned by $rpcparameters(pProcedureName), except that only the first 4 columns are required. See the section on Listing Remote Procedure Parameters ($rpcparameters) for details of the list layout.

The easiest way to define a procedure is to first call $rpcparameters(), fetch the result set into a list, and pass the list to $rpcdefine(). For example:

###### Do iResultsList.$define() Do StatementObj.$rpcparameters('byroyalty') Returns #F Do StatementObj.$fetch(iResultsList,9999) Returns lFetchStatus Do SessObj.$rpcdefine('byroyalty',iResultsList) Returns #F

Once a remote procedure has been defined, it can be invoked using the statement method $rpc().Use is of the form:

###### Do StatementObj.$rpc(pProcedureName,pParam1,…pParamN) Returns #F

The call to $rpc() will fail if pProcedureName has not been defined using $rpcdefine() or does not exist on the server. The session object will invoke the specified remote procedure, using the procedure definition to determine the parameters it needs. If the optional parameters pParam1…pParamN are included, they are passed to the stored procedure.

If the call is successful, any output parameter values are returned. If the procedure has a return value specified in its definition, it is written to the statement property $rpcreturnvalue, if not, $rpcreturnvalue is Null. If the call to the procedure generates a result set, the statement property $resultspending is set to kTrue and these results may be retrieved using the $fetch() method. Following successful execution of the remote procedure, the statement $state property will be set to kStatementStateExecDirect. If the state prior to this call was not kStatementStateClear, any pending result set or unexecuted SQL statement is cleared.

For example, to invoke the stored procedure “byroyalty” passing the variable lPercentage to the 1st parameter and fetch the results set generated by the stored procedure:

###### Do StatementObj.$rpc('byroyalty',lPercentage) Returns #F Do StatementObj.$fetch(iResultsList,9999) Returns lFetchStatus

## Transactions

The $transactionmode session property controls the way that transactions are managed. Depending on the value of this property the session object may automatically manage transactions or it may be necessary to manage transaction using explicit method calls or SQL statements.

Some servers do not provide support for transactions. You can determine whether a particular server allows transactions using the read-only Boolean session property $allowstransactions which contains kTrue if the server supports transactions or kFalse otherwise. Some session objects will contain a value of kUnknown in this property until the session is logged on. If your server does not allow transactions, the properties and methods described in the section on Transaction Modes below should not be used.

### Transaction Modes

You can set the $transactionmode session property using a command of the form:

###### Do SessObj.$transactionmode.$assign(kSessionTranManual) Returns #F

The potential values for the transaction mode are

1. **kSessionTranAutomatic**  
   This is the default and specifies that all transaction management be provided automatically.
2. **kSessionTranManual**  
   Enables the application to manage transactions manually using the session methods $begin(), $commit() and $rollback().
3. **kSessionTranServer**  
   Transaction management is provided by the DBMS.

#### Automatic Mode

After a SQL statement has successfully executed, i.e. $execute() or $execdirect() returns kTrue, the current transaction is automatically committed by the session object. If the command fails, the transaction is rolled back automatically. A new transaction is started automatically if required after a successful commit or rollback.

Note that since each individual SQL command is committed immediately that it is executed, automatic mode does not allow for the creation of transactions that contain a number of SQL commands to be individually prepared and executed prior to a single commit. If this is required, you should use Manual or Server mode.

#### Manual Mode

In kSessionTranManual mode you manage transactions manually using session methods.

**$begin()**

Where required by the session object, use the $begin()method to start a new transaction, for example:

###### Do SessObj.$begin() Returns #F

The $begin() method should only be executed where the DBMS does not implicitly start a transaction. The read-only session property $autobegintran will contain the value kTrue to indicate for a particular session object that a transaction is automatically started when a connection is established to the database or the previous transaction is committed or rolled back, and in this case the $begin() method should not be used. This method may fail if there is a current transaction or the server does not support nested transactions.

**$commit()**

Use the $commit()method to commit the current transaction, for example:

###### Do SessObj.$commit() Returns #F

The $commit() method will fail if the session $state property is kSessionLoggedOff, if the transaction mode is not kSessionTranManual or if there is no current transaction. With certain types of session objects this will commit and clear all statements, close any cursors used by a statement and clear pending results. This will not destroy the statement objects used by a session. Depending on the value of the session property $autobegintran, the server may begin a new transaction automatically.

**$rollback()**

Use the $rollback()method to roll back the current transaction, for example:

###### Do SessObj.$rollback() Returns #F

The $rollback() method will fail if the $state property is kSessionLoggedOff, if the transaction mode is not kSessionTranManual or if there is no current transaction. This method cancels the current transaction. With certain types of session objects this will rollback and clear all statements, close any cursors used by a statement and clear pending results. This will not destroy the statement objects used by a session. Depending on the value of the session property $autobegintran, the server may begin a new transaction automatically.

The read-only session properties $commitmode and $rollbackmode describe the effect that the $commit() and $rollback() methods have on statements and their cursors.

|  |  |
| --- | --- |
|  |  |
| kSessionCommitDelete kSessionRollbackDelete | Prepared statements and cursors on all statement objects are deleted. Any pending results are lost. The statement object itself is not deleted but will be set to a kStateClear state. To re-execute the same statement it must first be re-prepared. |
| kSessionCommitClose kSessionRollbackClose | Prepared statements and cursors on all statement objects are closed. Any pending results are lost. A statement can be re-executed without first being re-prepared. Any statement objects that have successfully prepared a statement will be in the kStatePrepared state. |
| kSessionCommitPreserve kSessionRollbackPreserve | The state of all statements and cursors remains unchanged. |

##### Note that with some session objects these properties may change once a session is logged on.

#### Server Mode

Transaction management is provided by the DBMS. The default behavior is determined by the database, which may for example automatically commit each SQL statement unless you override the default.

You may also execute SQL BEGIN, COMMIT and ROLLBACK or other statements depending on the DBMS SQL dialect, to manage transactions manually.

The read-only session property $autobegintran will contain the value kTrue to indicate for a particular session object that a transaction is automatically started when a connection is established to the database, or after a SQL COMMIT or ROLLBACK statement. If $autobegintran is kFalse, an explicit SQL BEGIN statement is required. SQL based transaction commands should not be used other than in kSessionTranServer mode.

As a general rule it is recommended that either automatic or manual mode should be used in preference to server mode.

In kSessionTranManual or kSessionTranServer mode the behavior of the DBMS dictates whether closing a connection commits the current transaction.

The effect of a commit or rollback on existing statement cursors is dependant on the behavior of the DBMS. In most cases, a commit or rollback will close all cursors in the session and clear all results sets. This does not destroy the statement object. It may be possible to re-execute the statement but generally the statement will need to be prepared again.

Care should be taken to note the circumstances in which commits occur as this can have a side effect on the processing of other statement objects associated with the session.

## Cursor Results Sets

When a statement is executed that generates results, the results set is preserved until another SQL command is executed. Cursor results sets enable you to process the results of 2 or more different SQL SELECT commands in parallel using a number of statement objects that were created from the same session object using the $newstatement() method.

Note: to use multiple concurrent cursors, the session transaction mode usually needs to be set to kSessionTranManual.

If the database does not implicitly allow for the concurrent processing of multiple results sets, you need to set the statement property $usecursor to a value of kTrue for each statement prior to executing the SELECT. This indicates that a statement should be created via a server-based cursor. It controls the server specific behavior of the $prepare(), $execute(), $execdirect() and $fetch() methods. In some circumstances, the client will automatically generate server-based cursors for SQL SELECT statements and therefore this property is ignored. If the session object manages cursors and the client does not support the manual generation of cursors, the session object will explicitly issue the SQL cursor commands.

Currently, DAMs which support the $usecursor property include the ODBC and Sybase DAMs. The Oracle and OmnisSQL DAMs provide $prepareforupdate() and $posupdate() methods, whilst DB2 provides its own record locking feature.

If the session object has to explicitly issue SQL cursor commands and a statement is prepared when $usecursor is kTrue, the following will be prefixed to the statement:

###### DECLARE <$name> CURSOR FOR <$sqlText>

It is important that $sqltext specifies a SQL SELECT statement. Note that the syntax of the DECLARE and associated SELECT command may vary slightly if a particular server does not adhere to the SQL-92 standard.

A subsequent $execute() and $fetch() will issue an

###### OPEN CURSOR <$name> and FETCH CURSOR <$name>

Depending on the value of $commitmode and $rollbackmode, any pending results set may be destroyed when a transaction is committed or rolled back.

To ensure that the results sets are not destroyed by an update command you need to set the transaction mode to kSessionTranManual and commit updates manually when ready using the $commit method.

For example, to create a new statement, execute and fetch results from a SELECT command using a cursor:

###### Do SessObj.$newstatement() Returns StatementObj Do StatementObj.$usecursor.$assign(kTrue) Do StatementObj.$execdirect('SELECT \* FROM authors ORDER BY au\_lname,au\_fname') Returns #F Calculate lFetchStatus as StatementObj.$fetch(iResultsList,10)

Non-Unicode Compatibility

The DAMs provided with Studio 5.0 are able to function in Unicode or 8-bit compatibility mode. This means that after converting your existing libraries for use with Studio 5.0, it should be possible to continue interacting with non-Unicode databases.

In 8-bit compatibility mode, all DAMs

* Return non-Unicode character data types via the $createnames() and $coltext attributes
* Bind outgoing character variables using the database’s non-Unicode data types
* Convert all data inside outgoing character bind variables to single-byte characters
* Define incoming character columns using the database’s non-Unicode data types
* Convert all data inside incoming character bind variables from bytes into the Omnis character set

### Switching to 8-bit compatibility mode

To switch to 8-bit compatibility mode, there is a session property: $unicode- which should be set to kFalse from its default value of kTrue. This implementation allows multiple Unicode and 8-bit session objects to exist side by side if required.

#### Character Mapping

This section is applicable to session objects operating in 8-bit compatibility mode only.

When reading data from a server database, Omnis expects the character set to be the same as that used in an Omnis data file. The Omnis character set is based on the macOS extended character set, but is standard ASCII up to character code 127. Beyond this value, the data could be in any number of different formats depending on the client software that was used to enter the data.

When assigned, the $maptable session property identifies files containing translation tables for character codes read into and sent out of Omnis. For example, suppose you are working with a database that stores EBCDIC characters. In order to accommodate this database, you should create an '.IN' map file that translates EBCDIC characters to ASCII characters when Omnis in reading server data and a matching '.OUT' file that reverses the process by converting ASCII to EBCDIC characters when Omnis is sending data to the server.

Under Windows and Linux, Omnis uses the same character set as under macOS, so in the general case, mixed platform Omnis applications should have no need for character mapping. However, if the data in a server table was created by another software package, running under Windows for example, the characters past ASCII code 127 would appear incorrect when read using Omnis. In this situation the $maptable property should be used to map the character set.

There are two kinds of character maps: IN and OUT files. IN files are used to translate characters coming from a server database into Omnis. OUT files are used to translate characters that travel from Omnis back to a server database.

#### The Character Map Editor

The Character map editor is accessed via the Add-On tools menu item and enables you to create character-mapping files. You can change a given character to another character by entering a numeric code for a new character. The column for the Server Character for both .IN and .OUT files may not actually represent what the character is on the server. This column is only provided as a guide. The Numeric value is the true representation in all cases.

To change a character, select a line in the list box and change the numeric code in the Server Code edit box. Once the change has been recorded, press the Update button to update the character map. You can increase/decrease the value in the Server Code edit box by pressing the button with the left and right arrows. Pressing the left arrow decreases the value, pressing the right arrow increases the value.

##### Make Inverse Map

The File menu lets you create new character map files, save, save as, and so on. The Make Inverse Map option creates the inverse of the current map, that is, it creates an “.IN” file if the current file is an ”.OUT” character map, and vice versa. When using the Make Inverse Map option, if any character is defined more than once within the IN.map then only the first value will be translated, and subsequent characters will be set to spaces (dec 32, hex 20). You should check the validity of the results when using the inverse mapping.

#### Using the Map Files

Establish the character mapping tables by setting the session property $maptable to the path of the two map files. Both files must have the same name but with the extensions .IN and .OUT and be located in the same folder. The $maptable property establishes both .IN and .OUT files at the same time. For example:

###### Do SessObj.$maptable.$assign('C:\Program Files\Tiger Logic\ Charmaps\pubs') Returns #F

In this example, the two map files are called “pubs.in” and “pubs.out”.

The session property $charmap controls the mode of character mapping that is to be applied to the data. Set the character mapping mode using a command of the form:

###### Do SessObj.$charmap.$assign(pCharMap) Returns #F

The potential values for the character mapping mode parameter pCharMap are

1. **kSessionCharMapOmnis**  
   Use the internal Omnis character set.
2. **kSessionCharMapNative**  
   This is the default and specifies that the client machine character set is to be used.
3. **kSessionCharMapTable**  
   Use the character mapping table specified in the $maptable property. If the $maptable property is not set and the application attempts to assign kSessionCharMapTable this fails.

If you wish to use the character mapping tables defined using the $maptable property, you must set $charmap to kSessionCharMapTable.

### Handling Extended Characters

When operating in non-Unicode mode, the session property $codepage determines how 8-bit character codes will be interpreted by the DAM. For example when $codepage is set to kUniTypeAnsiGreek, fetched ANSI extended character codes are interpreted as letters from the Greek alphabet. Conversely, Greek characters inserted from Omnis are mapped to character codes from the Greek code page.  
Thus, when operating in non-Unicode mode it is important that the value of $codepage matches with the character set being used by the remote database.

It should also be noted that when inserting data, any Unicode characters which do not correspond with characters in the selected code page will not be mapped correctly and such data is liable to loss or truncation.

Omnis character mapping is applied to fetched character data *before* conversion from the selected codepage, whilst inserted character data has Omnis character mapping applied *after* conversion to the selected code page.

### Interpreting 8-bit Data

This section is applicable to the MySQL, PostgreSQL and Openbase DAMs which interface with their respective client libraries using the UTF-8 encoding .

When operating in Unicode mode, it is possible to receive mixed 8-bit and Unicode data- since UTF-8 character codes 0x00 to 0x7F are identical to ASCII character codes.  
Where this data was created using the non-Unicode version of Omnis however, it is possible that the data may contain ASCII extended characters. In this case, the Unicode DAM will encounter decoding errors- mistaking the extended characters as UTF-8 encoded bytes.

This issue was not a concern for the non-Unicode version of Omnis Studio since extended characters were always read and written as bytes- irrespective of the database encoding.

In order to avoid problems when upgrading to the Unicode version of Omnis Studio, it is advisable to convert tables containing ASCII extended characters to UTF-8. This process is simplified where the database character set is already set to UTF-8 (as is often the case with MySQL). All that is required is to read and update each row in the table and repeat this for all tables used by the application. In so doing, Omnis will convert the 8-bit data to Unicode and then write the converted Unicode data back to the database.

In order to facilitate this within the DAM, the session property: $validateutf8 is provided. When set to kTrue (the default), any fetched character data is validated using the rules for UTF-8 encoding. Where a given text buffer fails validation, it is assumed to be non-Unicode data and is interpreted accordingly. When written back to the database, all character data will be converted to UTF-8. Such updates will result in frequently accessed records having their contents refreshed automatically.

By setting $validateutf8 to kFalse, validation is skipped and the DAM reverts to the previous behaviour- in which case extended ASCII characters should be avoided.

Aside from the issue of UTF-8 encoded data, the DAMs provided with Studio 5.0 are able to retrieve non-Unicode data from non-Unicode database columns in either Unicode or 8-bit compatibility mode. The DAM knows the text capabilities of each character data type and assigns encoding values to each result column accordingly.

The difference in behaviour when using 8-bit compatibility is that in compatibility mode, it is also possible to write data back to non-Unicode columns.

In Unicode mode, the DAM assumes that it will be writing to Unicode compatible data types and this will cause data insertion/encoding mismatch errors if the clientware tries to insert into non-Unicode database columns.

#### Character Mapping in Unicode Mode

Character mapping to and from the Omnis character set is also possible where session objects are operating in Unicode mode. This was previously removed from the Unicode DAMs since it provided compatibility between the various 8-bit character sets. Where Unicode DAMs encounter 8-bit data however, the session $codepage property indicates the ANSI code page which should be used to interpret the data.  
In the general case, you cannot insert non-Unicode data when the DAM is operating in Unicode mode. To insert such data you should switch to 8-bit compatibility mode ( by assigning $unicode to kFalse).

### Server Specific Programming

Certain DAMs, namely DAMORA8 and DAMODBC also provide session properties which allow mixing of Unicode and 8-bit data when the DAM is operating in Unicode mode.

The Oracle DAM provides $nationaltonvarchar and $nationaltonclob which allows the Omnis *National* character subtype to be used with Unicode data, whilst the *Character* subtype is reserved for non-Unicode data.

The ODBC DAM provides $nationaltowchar which performs a similar function. These properties are documented further in the Server Specific Programming chapter.

The onus is upon the developer not to put Unicode characters into Character subtypes when using these properties, otherwise data insertion/encoding mismatch errors will occur.

## Stripping Spaces

The session object can automatically strip trailing spaces from data returned from the server. This functionality is switched on by setting the statement $sqlstripspaces property to kTrue. The default value for a statement is taken from the session object $sqlstripspaces and the default for the session is read from $clib.$prefs.$sqlstripspaces.

Some data sources may strip trailing spaces prior to sending it to the session object in which case this property has no effect.

## Treatment of Date Values

The way the session handles partial, empty and NULL valued date columns can be modified using two properties; sessionObj.$defaultdate and sessionObj.$emptydateisnull.

The $defaultdate property is used to specify default date parts to be used when parts of the date are omitted at bind time. Parts of the date (day, month and/or year) are also substituted where the value being inserted would otherwise constitute an invalid date. This property provides better support for Omnis custom date types, for example DateTime ‘H:N:S.s’ which may have a corresponding server type which includes the date. Bind variable values override default date values for the constituent date parts which are supplied. The default value for this property is “1 Jan 2000 00:00:00”. It is not possible to assign a #NULL value to $defaultdate.

$emptydateisnull can be used to set all outgoing bound dates with an empty value to NULL. This applies to date bind variables used in the WHERE clauses of SELECT, UPDATE and DELETE statements as well as to the input bind variables of INSERT and UPDATE statements. To insert (or test for) empty date values, it is necessary to set $emptydateisnull to kFalse and to assign an empty date to $defaultdate.

The implications of these two properties are summarised below:

|  |  |  |  |
| --- | --- | --- | --- |
| To… | Use value… | $defaultdate | $emptydateisnull |
| INSERT a NULL date into a datetime column or  SELECT…WHERE a date value is NULL | #NULL or <empty> | Ignored  Ignored | Ignored  kTrue |
| INSERT an <empty> date value into a datetime column or  SELECT…WHERE a date value is <empty> | <empty> | <empty> | kFalse |
| INSERT/test for a date in a datetime column, substituting a default time, (empty datetimes are treated as NULL) or  INSERT/test for a Time in a datetime column, substituting a default date, (empty datetimes are treated as NULL). | Datetime value | Default datetime | kTrue |
| INSERT/test for a Date in a datetime column, substituting a default time, (empty datetimes take on $defaultdate) or  INSERT/test for a Time in a datetime column, substituting a default date, (empty datetimes take on $defaultdate). | Datetime value | Default datetime | kFalse |

## Large Objects

When working with large objects (LOBs) there are some properties that may be used to tune the system so that data is handled and transferred in an efficient way. Large objects include large text, list, binary, picture data and instances of object classes.

### Blob Size

The session property $blobsize defines the maximum size of a binary data item. The default value for this property is 10MB, but it may be adjusted to conserve memory. Some session objects may take this value into account when executing a $createnames() method and resulting data types may vary according to the value of $blobsize.

### Chunking Large Objects

In order to pass large objects to the server they are broken down into chunks. This is due to potential memory limits that exist with some database vendor client APIs. The session properties $lobchunksize and $lobthreshold represent respectively the size of a single chunk in bytes and the size of object in bytes at which chunking starts. The default value of both of these properties is 32KB.

This only applies to Omnis Character, Picture, Binary, List and Object data types that are greater than 255 bytes.

The value of $lobthreshold may be set to any value between 256 bytes and 2GB. Note that if not set judiciously, this may cause resource problems due to the amount of memory required to cache an object.

The value of $lobchunksize may be set to any value between 256 and the $lobthreshold.

Due to limitations in the database vendor client API, certain session objects may impose their own values for the default and maximum chunk sizes.

## Session Pools

Pools of session instances can be created when Omnis starts up and made available to any methods running within Omnis. They are designed to be used by the multi-threaded server and allow client methods to quickly obtain SQL sessions without the burden of constructing their own instances.

There is a notation group $root.$sessionpools that contains the current session pools. Normally the session pools are created by the startup task and exist until Omnis is shut down. Pooled sessions do not appear in the SQL Browser.

When a session is required in order to perform a SQL query, it is obtained from the pool using an object variable and a statement object created in the normal way in order to execute the query and fetch any results. When the session object variable is destroyed the session instance is returned to the pool for later reuse.

### Creating a Session Pool

Session pools are created using the $makepool() session object method. The syntax of the method is:

###### $makepool(name[,count=0,hostname,username,password,initmethod]) # creates a pool of session objects

The call to $makepool() will only be successful if used with an external object that is a session object or an object class that has a session object as its subtype, that is, the external server object is its superclass. The method is executed for each session being created before the session is actually logged on.

The $makepool() method returns a reference to the pool if it is successfully created and the required number of session instances are constructed, otherwise it returns NULL. Once a session pool has been created there is no need to maintain it further during the time that the library is open, however it is possible to change the number of available instances using notation.

Errors encountered when creating session pools are returned via #ERRCODE and #ERRTEXT.

#### Using $makepool() with an External Object

Create a session pool using an external object with a method call of the form:

###### Do $extobjects.DAMobject.$objects.SessObject.$makepool(pPoolName, pCount,pHostName,pUserName,pPassword) Returns lPoolRef

DAMobject is the name of the external component. You can find out what DAM objects are available on your workstation by examining the $root.$extobjects branch of the notation tree using the Notation Inspector. SessObject is the name of the session object. You can find out what session object is available by expanding the $objects group for a particular DAM object using the Notation Inspector. The pPoolName parameter is the name of the pool and must be unique amongst session pools and the pCount parameter is the number of object instances to be initially contained in the pool. The other parameters are optional and if specified are passed to the $logon() method for the instance. If they are not specified, the instance is constructed but not logged on.

For example, to create a session pool called “poolone” containing 5 sessions all logged on to SQL Server

###### Calculate lHostName as 'SqlServer' Calculate lUserName as '' Calculate lPassword as '' Do $extobjects.ODBCDAM.$objects.ODBCSESS.$makepool('poolone',5, lHostName,lUserName,lPassword) Returns lPoolRef

#### Using $makepool() with an Object Class

Alternatively $makepool() may be used with an object class with a method call of the form

###### Do $objects.ObjectClass.$makepool( pName,pCount,pHostname, pUserName,pPassword) Returns lPoolRef

ObjectClass is the name of an object class that must have a session object as its superclass. You can achieve this by selecting the object class in the browser and clicking on the $superclass in the Property Manager, click the arrow and select External Objects and double-click on the required session object.

For example:

###### Do $objects.odbcobj.$makepool('pooltwo',10,lHostName,lUserName, lPassword) Returns lPoolRef

#### Initialising session objects

The makepool() method has a sixth parameter that allows you to pass the name of an initialisation method to the session object in the form ‘class.method’. The initialisation method needs to have a parameter at position 1 of Object Reference type. This method is called for each session added to the pool. The Object Reference Parameter will contain a reference to the newly created session which can be used to initialise the session object. Example:

###### Do $extobjects.MYSQLDAM.$objects.MYSQLSESS.$makepool('pool1', 5,'192.168.1.25', 'user1', 'mypass', 'NewWindow.$myinit') Returns lPoolRef

### Obtaining a Session Instance From a Pool

The $new() method is used to assign a session instance from a pool. For example

###### Calculate SessObj as $sessionpools.poolone.$new()

If $new() is successful the session instance assigned from the pool belongs to SessObj and is normally returned to the pool when SessObj is destroyed (for example, if SessObj is a local variable the session is returned to the pool when the method containing SessObj terminates). Alternatively the session can be manually returned to the pool by assigning some other object or zero to SessObj. The $new method returns NULL if all instances contained in the pool have already been assigned out.

Now you can use the $newstatement() method with the session object to create a statement to execute SQL in the normal way.

### Session Pool Notation

The group $sessionpools supports the usual $findname(), $first(), $next() and $makelist() notation. The $remove() method is also implemented but not $add() since $makepool() is used to create a session pool.

A pool has the $name, $poolsize and $inuse properties. The $poolsize property is the number of instances stored in the pool and $inuse is the number of these which are currently assigned out. If you increase the value of $poolsize the new session instances are immediately constructed and if the hostname, username and password parameters were specified at $makepool, they are also logged on.

You can use poolRef.$poolsize.$assign(poolRef.$poolsize-1) to reduce the pool size, that is, to destroy a session instance in the pool, providing enough sessions are available to be destroyed. If not, then the pool size is reduced as and when sessions are returned to the pool. Note that the $poolsize property reflects the actual number of active sessions and not necessarily the number of sessions required by the user.

An alternative form of the $new() method is poolone.$new(pWaitSeconds) which is designed to be used in client methods running on the multi-threaded server. If there are no sessions currently available in the pool this waits for the specified number of seconds. Whilst it is waiting other threads are allowed to run and if a session is returned to the pool it will be used by the waiting method. At the end of the time period NULL is returned if no session has become available. Note that this waiting time period should be treated as approximate.

### Destroying a Session Pool

A session pool normally exists for the lifetime of the Omnis process. You can forcibly destroy a session pool using notation, passing an item reference to the $sessionpools.$remove() method. For example:

###### Set reference ref to $sessionpools.pool1.$ref() Do $sessionpools.$remove(ref) Returns #F

## Diagnosing Problems

As of Omnis Studio 5.0, the session object provides the $debugfile and $debuglevel properties. These are useful in the event of program faults, if you are looking to optimise network traffic or if you are simply curious about how the DAM is executing commands. It should be noted that when debugging is enabled, there is a noticeable impact on performance. Hence, debugging should be reserved for application development and technical support issues- in which case use of $debuglevel 4 is recommended.

When $debugfile is set to a valid filename, the DAM starts writing debug information to this file- which is cleared before use. The file is created if it does not already exist. Debugging continues either until the session object is destructed or until $debugfile is set to an empty string. For macOS and Linux, a POSIX style path is expected and the user must have permission to write to the file at the specified location.

It is also possible to assign the value “stderr” to $debugfile. This is useful for macOS and Linux platforms where Omnis is run from the command prompt. Debug information will be written to the terminal window.

The $debuglevel property determines the level of debugging information that is written to the debug file and supports the following values:

|  |  |
| --- | --- |
| 0 | No debugging. The debug file remains open but debugging output is suspended until $debuglevel is set to a higher level. |
| 1 | Base level debugging. At this level, the DAM base class writes high level descriptions about the operation of the DAM; statement prepares, executes, describing of result sets, fetching, etc. This is the default level of debugging. |
| 2 | Detail refinement 1. At this level metadata, property and method call information are also written including the SQL text associated with $prepare()s and $execdirect()s. |
| 3 | Detail refinement 2. At this level, buffer allocation, parameter and bind variable values are also written where possible. |
| 4 | Detail refinement 3. At this level, details of parameters passed to implementation API calls are also written, provided that the DAM implements this level of debugging. |
| 5 | Causes a time stamp to be prepended on to each debug entry. The time stamp is accurate to 1/60th second and reflects the time since the session object logged-on. Debug lines written before $logon() reflect the system up-time. |

## Session and Statement Properties and Methods

Below is a summary of the base methods and properties common to all object DAMs.   
For details of DAM specific methods and properties, refer to the chapter on *Server Specific Programming.*

Session Methods

|  |  |
| --- | --- |
| Method | Description |
| $begin() | $begin() explicitly begins a new database transaction. |
| $clear() | $clear() clears all statement objects based in the session and resets the session to its default state. |
| $coltext | $coltext(vVarRef) returns the DBMS specific SQL text corresponding to the data-type of the supplied variable. |
| $commit() | $commit() explicitly commits the current database transaction. |
| $createnames() | $createnames(vTableDef,[bNullInfo,bPrimaryKeyInfo]) returns a DBMS specific text string consisting of the SQL column names and types based on the column types of the supplied list or row variable. The optional parameters bNullInfo and bPrimaryKeyInfo can be used to request additional information about columns where the list has been defined from a schema or file class. |
| $definelistorrow() | $definelistorrow(&vListOrRow,cTableName) defines a list or row from the specified server table. (Studio 8.1.5 and later) |
| $insertnames() | $insertnames(wRowRef) returns a text string consisting the SQL column names and bind variable names corresponding to the supplied list or row variable. |
| $logoff() | $logoff() disconnects the session from the database. |
| $logon() | $logon(cHostname,cUsername,cPassword[,cSessionName]) connects the session to the DBMS using the supplied parameters, The optional cSessionName parameter registers the session with $root.$sessions and the SQL Browser utility. |
| $makeschema() | $makeschema(pSchema,cTableName) makes a schema class based on the specified server table name. |
| $newstatement() | $newstatement([cStatementname]) returns an instance to a new statement object. cStatementname is optional. |
| $newstatementref() | $newstatementref([cStatementname]) returns a reference to a new statement object. cStatementname is optional. |
| $nextnativeerror() | $nextnativeerror() retrieves a pending DBMS error code and error message, placing them in $nativeerrorcode and $nativeerrortext. |
| $rollback() | $rollback() explicitly rolls-back the current database transaction. |
| $rpcdefine() | $rpcdefine(cProcedureName,lParamList) defines the parameter structure for a subsequent call to statement.$rpc(). The definition and contents of the lParamList parameter are discussed in *Calling a Remote Procedure* |
| $selectnames() | $selectnames(vTableDef[,cTableName]) returns a text string consisting of comma delimited column names corresponding to vTableDef. The optional cTableName is pre-pended to the column names if supplied. |
| $updatenames() | $updatenames(wRowRef) returns the text string for a SQL UPDATE clause based on the contents of wRowRef. |
| $wherenames() | $wherenames(wRowRef[,cTableName,cComparison,cOperator]) returns the text string for a SQL WHERE clause based on the supplied list or row variable. The optional cTableName is pre-pended to the column names if supplied. The cComparison and cOperator parameters can be used to modify the corresponding parts of the WHERE clause. |

Session Properties

|  |  |
| --- | --- |
| Property | Description |
| $allowstransactions | kTrue if the session is capable of manual transactions. (Read-only) |
| $apiversion | Version of the database client API that the DAM was built with. (Read-only) |
| $autobegintran | kTrue if the session automatically begins transactions, e.g. each time a transaction is committed/rolled-back. In which case, $begin() should not be used. (Read-only) |
| $batchsize | The desired batch size used when fetching multiple rows in a single network transaction. Statement objects inherit this value, allowing its use with SQL worker objects. Default value: 1. (Studio 10.0.1) |
| $blobsize | The maximum size of a binary data item. Default value 10MB. Also used by $createnames() for some DBMSs. |
| $charmap | Determines how character data received from and sent to the DBMS is mapped. For Non-Unicode session objects, this property accepts values of kSessionCharMapOmnis (the default), kSessionCharMapNative or kSessionCharMapTable.  For Unicode session objects, this property governs the character set assumed for fetched 8-bit data; either kSessionCharMapLatin or kSessionCharMapRoman. |
| $commitmode | Indicates how SQL cursors behave when a session is committed. Either kSessionCommitClose, kSessionCommitDelete or kSessionCommitPreserve. (Read-only) |
| $damname | The DAM name as shown in $root.$components. (Read-only) |
| $debugfile | When set to a valid filename, the DAM starts writing debug information to this file. Debugging continues either until the session object is destructed or until $debugfile is set to an empty string. For macOS and Linux, a POSIX style path is expected. |
| $debuglevel | Determines the level of debugging information that is written to the $debugfile. 0 specifies no debugging, 4 specifies maximum debugging info. |
| $defaultdate | Used to specify the default date parts to be used when parts of the date are omitted at bind time. The default value for this property is “1 Jan 2000 00:00:00” |
| $emptydateisnull | If set to kTrue, all out-going bound dates with an empty value will be set to NULL. |
| $encoding | Indicates the Unicode encoding used by the client library for bind data and fetched rows. The DAM converts to and from this encoding when exchanging data with Omnis. (Read-only) |
| $codepage | Set to one of the kUniType… constants found in the Catalog under *Unicode types*. Default value kUniTypeAnsiLatin1. This property determines the ANSI code page used to interpret non-Unicode data. $codepage should match the character set of the database so that non-Unicode extended characters are read and written correctly. |
| $errorcode | Returns the Internal Omnis error code generated by last executed session method. (Read-only) |
| $errortext | Returns the error message generated by the last executed session method. (Read-only) |
| $fetch64bitints | If kTrue (default), 64-bit integers are fetched into 64-bit Integer fields. If kFalse, they are fetched as 32-bit Integers and truncated accordingly. This property provides backward compatibility with the old-style web client plug-in which does not support 64-bit integers. |
| $hostname | The hostname currently in use by the connection. (read-only) |
| $lobchunksize | The size (in bytes) of a single chunk when inserting large binary data. Default value is 32KB |
| $lobthreshold | The size (in bytes) of a binary data item at or above which chunking will occur. Default value is 32KB |
| $maptable | The path to the character map files to use when $charmap is assigned kSessionCharMapTable. The filename is specified without either the “.in” or “.out” suffix. |
| $nativeerrorcode | The error code generated by the DBMS or DBMS clientware in response to the last executed session method. (Read-only) |
| $nativeerrorpending | kTrue indicates that a further native error code and error text are available. See session.$nextnativeerror().(Read-only) |
| $nativeerrortext | The error message generated by the DBMS or DBMS clientware in response to the last executed session method. (Read-only) |
| $password | The password currently in use by the connection. (Read-only) |
| $quotedidentifier | If kTrue, table and column name identifiers returned from the $createnames(), $insertnames(), $updatenames(), $selectnames() and $wherenames() methods will be quoted “thus”, facilitating case-sensitive names and names containing spaces. Affects table instance methods as well as session object methods. (Studio 5.2 and later) |
| $rollbackmode | Indicates how SQL cursors behave when a session is rolled-back. Either kSessionRollbackClose, kSessionRollbackDelete or kSessionRollbackPreserve. (Read-only) |
| $sqldecimalseperator | The character that the DBMS uses to represent the decimal separator when storing numeric values, usually ‘.’ |
| $sqlstripspaces | If kTrue, trailing blank characters are stripped from character data returned from the DBMS. |
| $sqlthousandsseperator | The character that the DBMS uses to represent the thousands separator when storing numeric values, usually ‘,’ |
| $state | Indicates the current state of the session object’s connection. Either kSessionStateLoggedOff or kSessionStateLoggedOn. (Read-only) |
| $transactionmode | Used to set the transaction mode. Can be one of kSessionTranAutomatic (the default), kSessionTranManual or kSessionTranServer. |
| $unicode | Used to enable/disable 8-bit compatibility mode. If kTrue (the default), all character data is exchanged with the client as Unicode. If kFalse, the behaviour of a non-Unicode DAM is adopted. |
| $username | The username currently in use by the connection. (Read-only) |
| $validateutf8 | If kTrue and the $encoding is kSessionEncodingUtf8 and $unicode is kTrue, fetched UTF-8 data is validated and treated as non-Unicode data on failure. See also; $charmap. Not implemented by all DAMs. |
| $version | Once a session has been established this is the version of the database the object is connected to. This defaults after a $logoff(). (Read-only) |

Statement Methods

|  |  |
| --- | --- |
| Method | Description |
| $clear() | $clear() clears pending results and resets the statement. |
| $columns() | $columns(cTableName[,iFlags]) generates a result set describing the columns of the specified table. An optional flags parameter can be specified to generate column lengths for Number, Integer and Date columns. Values to be ORed together can be found in the catalog under Statement Flags. |
| $execdirect() | $execdirect([cSqlText]) directly executes the specified SQL text or executes the contents of the Statement buffer if cSqlText is omitted. |
| $execute() | $execute() executes previously prepared SQL text. |
| $fetch() | $fetch([lListOrRow,iRowCount,bAppend]) fetches the specified number of rows from the result set into the supplied list variable. If iRowCount is omitted, a single row is fetched. If bAppend is kFalse or omitted, the list or row contents are cleared prior to the fetch. |
| $fetchinto() | $fetchinto(vParam1…vParamN) fetches one row of the result set and stores each column in the supplied variables, one variable for each column. |
| $fetchtofile() | $fetchtofile(cFileName[,iRowCount,bAppend,bColumnNames]) fetches the specified number of rows from the result set and stores them in the specified file. iRowCount, bAppend and bColumnNames parameters are optional. |
| $indexes() | $indexes(cTableName[,iIndexType]) generates a result set providing information on the indexes of the specified table. The optional iIndexType can be one of kStatementIndexUnique (default), kStatementIndexNonUnique or kStatementIndexAll. |
| $nextnativeerror() | $nextnativeerror() retrieves a pending DBMS error code and error message for the statement object, placing them in $nativeerrorcode and $nativeerrortext. |
| $prepare() | $prepare([cSqlText]) prepares the supplied SQL text ready for subsequent execution. If omitted, the contents of the statement buffer are prepared instead. |
| $results() | $results(lListOrRow) populates the supplied list variable with a description of the columns contained in the current result set. |
| $rpc() | $rpc(cRpcName,vParam1…vParamN) calls the specified remote procedure. Any supplied parameters are passed to the procedure call. |
| $rpcparameters() | $rpcparameters(cRpcName) generates a result set describing the parameters used by the specified remote procedure. |
| $rpcprocedures() | $rpcprocedures([cOwnerName]) generates a result set containing the names of remote procedures callable by the specified user. If omitted, all procedure names are returned. |
| $tables() | $tables([iTableType,cTableOwner]) generates a result set containing the names of tables accessible by the specified user. If cTableOwner is omitted all tables are returned. iTableType can be one of kStatementServerTable, kStatementServerView or kStatementServerAll (the default). |

Statement Properties

|  |  |
| --- | --- |
| Property | Description |
| $batchsize | The number of simultaneous rows to be retrieved for a single network transaction. Defaults to 1\* but accepts any long integer value. Not implemented by all DBMSs, in which case this property will be read-only. (\*In Studio 10.0.1, $batchsize is inherited from sessionObject.$batchsize). |
| $effectivebatchsize | Reflects the maximum attainable batchsize for the current statement. Will be less than $batchsize if memory for the desired batch size cannot be accommodated. (equivalent to $maxbuffersize / largest-column-size) (Read-only) |
| $columncount | The number of columns contained in the current result set. (Read-only) |
| $columndelimiter | The character used to delimit column values when fetching data using $fetchtofile(). Defaults to kTab but accepts any single character value. Non-printable characters should be assigned using the chr() function. |
| $errorcode | The Omnis internal error code generated by the last statement method. (Read-only) |
| $errortext | The error message generated by the last statement method. (Read-only) |
| $maxbuffersize | Used in conjunction with $batchsize. Sets the maximum buffer size used to store an array of column values when batch fetching. Defaults to 32KB but accepts any value larger than 255. Not implemented by all DBMSs, in which case this property will be read-only. |
| $maxresultsetsize | Designed to prevent Omnis from running-out of memory during $fetch() operations, this property limits the size of a result set to the specified value. The default value is 100MB |
| $nativeerrorcode | Error code generated by the last statement method. (Read-only) |
| $nativeerrorpending | Indicates that a further error message is available. Use $nextnativeerror() to retrieve. (Read-only) |
| $nativeerrortext | Error message generated by the last statement method. (Read-only) |
| $resultspending | Indicates that the last statement method generated some results or that there is another result set pending. (Read-only) |
| $rowcount | The number of rows in the current result set. If a particular session object cannot determine this value, this property returns –1. (Read-only) |
| $rowdelimiter | The character used to delimit row values when fetching data using $fetchtofile(). Defaults to kCr but accepts any single character value. Non-printable characters should be assigned using the chr() function. |
| $rowsaffected | The number of rows affected by the last executed statement; usually an INSERT, UPDATE or DELETE statement. If a particular session object cannot determine this value, this property returns –1. (Read-only) |
| $rowsfetched | The number of rows retrieved by the last fetch method to be executed. (Read-only) |
| $rpcreturnvalue | The return value of the most recently executed call to $rpc().(Read-only) |
| $state | A constant value indicating the current state of the statement object. (Read-only) |
| $statementname | The name which was assigned to the statement object during creation. (Read-only) |
| $sqlstripspaces | Denotes that fetched character data should have trailing spaces stripped. |
| $sqltext | The DBMS representation of the last SQL statement submitted to $prepare() or $execdirect(). (Read-only) |
| $usecursor | Denotes that this statement object should be associated with a SQL cursor. |

## SQL Multi-tasking and SQL Workers

You can execute long-running tasks such as a SELECT statement on a separate background thread that reports back to the main thread as each task completes. To enable this functionality, the Omnis DAMs allow the creation of “SQL Workers” which are instantiated from a new SQL Object variable type available in the Oracle, ODBC, MySQL, PostgreSQL, DB2, Sybase, and SQLite DAMs.

SQL Worker object completion methods allow list fields and other form data to be populated asynchronously, making applications more responsive and potentially faster where multiple SQL Workers are used simultaneously. There is an example library in the **Hub** (available when you start Omnis Studio) showing how you can use the SQL Worker Objects: the example connects to a SQLite database and runs multiple queries, some with bind variables, running at the same time.

### Overview

The SQL Worker Objects support three primary methods:

1. **$init()**   
   Initialises or resets a worker object ready to perform its task
2. **$start()**   
   Starts the worker task on a background thread (non-blocking); can be called multiple times to run different threads simultaneously
3. **$cancel()**   
   Aborts a worker task running on a background thread

There are additional properties to allow a running task to be discarded in place of a new task and to cancel such tasks as they become “orphaned”. There is also a property to report the state of a worker object's running background thread.

Worker objects are created by sub-classing an Omnis Object class with the appropriate SQL Worker Object type. You initialise the object by supplying a SQL statement along with any bind variables that the SQL statement may need. Logon details or optionally the name of a session pool are also passed during initialisation.

A SQL Worker thread is dispatched by calling $start(). Upon completion, the worker thread calls back into the worker object's $completed() method, or $cancelled(), with the result set or error information.

### SQL Worker Object Methods

|  |  |
| --- | --- |
| Method | Description |
| $init() | $init(ParamRow). Initialises or resets a worker object ready to perform a unit of work.\* |
| $start() | Starts the worker task running on a background thread (non-blocking).\* |
| $run() | Starts the worker task running on the caller thread (blocks until complete). Intended for testing purposes.\* |
| $cancel() | Aborts a worker task running on a background thread.\* |
| $sessionref() | $sessionref(ObjectRef). Returns a reference to the session object being used by the underlying background thread.\* |
| $completed() | Called by the background thread upon completion of its work. |
| $cancelled() | Called by the background thread if the running background task was cancelled. |

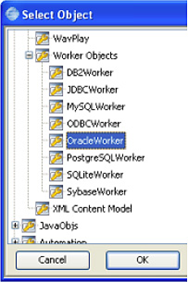
##### \*Returns kTrue on successful execution, kFalse otherwise.

### SQL Worker Object Properties

|  |  |
| --- | --- |
| Property | Description |
| $cancelifrunning | If kFalse (the default), orphaned background tasks run to completion. If kTrue, they are instructed to cancel before being detached. |
| $waitforcomplete | If kTrue (the default), the Interface Object waits for completion of a running background task before the object can be used again. If kFalse, the running task is detached and a new background thread takes its place. |
| $state | Returns the current state of the underlying background task; either kWorkerStateCancelled, kWorkerStateClear, kWorkerStateComplete, kWorkerStateInit or kWorkerStateRunning. |
| $errorcode | On failure of a command function, contains the error code. |
| $errortext | On failure of a command function, contains the error message. |
| $threadcount | Reports the number of active threads spawned by the worker object, including detached threads. |

### Creating SQL Worker Objects

Worker objects are created by sub-classing an Omnis object class as a Worker Object. For example, using the Select Object dialog, assigning a $superclass for use with an Oracle connection results in: ***.ORACLE8DAM.Worker Objects\OracleWorker***.



To access worker functionality from your code, you then create one or more object instance variables of subtype <your-object >.

#### Worker Object Initialisation

A worker object must be initialised on the caller thread before it can run. You initialise the object by supplying a SQL statement along with any bind variables that the SQL statement may require. Logon details, or optionally the name of a session pool, are also passed during initialisation.

The initialisation parameters are supplied to the $init() method via a row containing attribute values. Attribute names appear in the column headings. The attribute names recognised by the Worker Object are as follows (case-insensitive):

|  |  |
| --- | --- |
| Attribute name | Attribute value |
| session | A session object or object reference. The session must be logged-on and in a useable state. |
| poolname | The name of an existing session pool. The worker will take a session object from this pool, returning it upon completion. |
| hostname | The hostname/IP address of the database server. |
| database | The database name to use for a logon. |
| username | The username to use for a logon. |
| password | The password to use for a logon. |
| query | The SQL statement to be executed by the worker object. |
| bindvars | A list containing bind variable values. Bind variables are matched by name. If the list contains multiple rows, the query is re-executed for each row. |
| work | A list containing multiple SQL queries and associated bind variables. If specified in place of *query* and *bindvars*, allows the worker object to execute multiple SQL statements and return any result sets associated with each query. |

If the *session* attribute is supplied, the other logon attributes, i.e. hostname, database, username & password are ignored, since it is assumed that the session object is already in a useable state.   
**Please Note:** In this mode, the session should be considered reserved for use exclusively by the worker. If the main application attempts to share a session object being used by a worker running on another thread, the results are undefined.

The logon parameters are also ignored if the poolname attribute is supplied. In this mode, the worker attempts to obtain a session from the named session pool, releasing it when the worker task completes. (If both session and poolname are supplied, poolname is ignored.)

Where neither, session or poolname are supplied, an internal session object is created dynamically. Valid logon credentials should be supplied via hostname, username and password. Although read during the call to $init(), the worker will not attempt to logon until the $run() or $start() method is called. In this mode, the session is automatically logged-off when the worker task completes (or is cancelled). Should you need to modify one or more session attributes before calling $run() or $start(), it is possible to obtain a reference to the session object by calling the worker object’s $sessionref() method, for example:

###### Do iWorkerObj.$sessionref(lObjRef) Returns #F Do lObjRef.$port.$assign(5435)

The SQL text supplied via the query attribute may contain any SQL statement but ideally, should be a statement that normally takes an appreciable amount of time to execute, for example; a SELECT, UPDATE or DELETE statement. The query text may also contain one or more bind variables, specified using standard @[…] notation.

Bind variable values are supplied via a separate *bindvars* parameter. The supplied list is stored during $init() and read when the worker task starts. Where the list contains multiple rows, the worker re-executes the supplied SQL statement for each row of bind variables. Bind variable place holders in the query must reference columns by name within the bindvars list.

#### Running the Worker Object

The $start() method causes the worker task to run on a background thread. Thus, return from $start() is immediate and the main application is free to continue processing. For example, the iWorkerObj var has been created from the oPostgreSQLWorker class:



###### # iWorkerObj is an instance of an Object class Calculate Params as row(iSQLText,'192.168. 0.10’,iUser,iPassword,iDBName) Do Params.$redefine(query,hostname,username,password,database)

###### Do iWorkerObj.$init(Params) Do iWorkerObj.$start() Returns #F

The $run() method is analogous to $start() but provided for debugging and testing purposes only. In this mode, the benefit of the worker object is negated owing to the fact that the worker will run on the same thread as the caller, thus blocking the caller thread until the worker task is complete.

Once initialised, a worker object may be run repeatedly if desired provided that the supplied session object remains useable, the session pool has one or more free sessions or that the logon credentials remain valid. Any bind variables supplied will be re-used each time the worker is run.

If an error occurs during $init(), $start() or $run(), an error message is returned via the object’s $errorcode and $errortext properties.

#### Processing Worker Results

When complete, the worker task causes the main thread to jump into one of the worker object’s *callback* methods:

1. $completed()   
   This method is called with a row parameter defined with two columns: Results: a three-column list containing zero or more SQL result sets (lists) together with their associated QueryNum and BindRow where appropriate. Errors: a four-column list containing ErrorCode, ErrorMsg, NativeErrorCode and NativeErrorMsg values.
2. $cancelled()   
   This method is called (without parameters) if the user calls $cancel() on the worker object whilst it is running. When cancelled, any pending results are discarded.

A library may contain multiple worker objects of a given type. Each may be assigned a separate unit of work (SQL query) and each may be started asynchronously. It is the responsibility of the completion method in each worker object to process its result information and make this available to the main application as/when it becomes available. For example, here is a $completed() method:

###### Calculate List as pRow.Results.1.1 ##extract the first result set Calculate Errors as pRow.Errors ##extract list containing error info If Errors.$linecount()>0   Do method reportError(Errors) Else   Do method updateWindow(List) ## see comment below End If Calculate iThreadCount as $cinst.$threadCount ##shows how many threads are running

If the results returned by $completed() are to be displayed in a window or a remote form in the JavaScript Client, you will have to explicitly redraw the window instance, or in the case of a web form the remote client must contact the server to get updated automatically.

#### Executing Multiple SQL Statements

If the worker object is initialised with a *work* parameter in place of the *query* and *bindvars* parameters then a list of SQL statements will be executed and the worker’s $completed() method will be called once all statements have been executed and any result sets have been generated. The work parameter is a list defined with two columns;

* query – a character column containing the SQL statement plus any required bind variable place holders
* bindvars – a list containing one or more rows. Bind variable place holders in the query must reference columns by name within the bindvars list

**Example**

###### Do work.$definefromsqlclass('scWork') ## scWork is defined as query(char) and bindvars(list) Do work.$add('select \* from table1 where col1 < @[binds1.col1]',binds1) Do work.$add('select count(\*) from table1',) Do work.$add('select \* from table2 where idCol >= @[binds2.idCol]',binds2) Do work.$add('select oneCol from table3',)

###### Do initRow.$definefromsqlclass('scWorkInit') ## scWorkInit is defined as hostname(char), username(char), password(char) & work(list) Do initRow.$add(lHostname,lUsername,lPassword,work)

###### Do iWorker.$init(initRow) Returns #F Do iWorker.$start() Returns #F

Note that when the worker object’s $completed() method is called, the user is responsible for associating each result set with its originating SQL query, e.g.

###### Calculate ResultInfo as pResults.Results.1 Calculate Result1 as ResultInfo.Result

#### Manual Transaction Mode

If the session object passed to the worker is placed in manual transaction mode (kSessionTranManual), then all SQL statements executed by the worker object are executed as a single transaction which will either be committed on completion of the last statement, or rolled-back if one of the statements returns an error.  
To use manual transaction mode, the worker object should be supplied with a pre-initialised session object, using the *poolname* or *session* parameters.  
Manual transaction mode has no effect for SELECT statements, although if specified as part of a *work* list, the SELECT will not be executed if an error occurs in one of the preceding statements.

#### Worker State

The current state of a SQL Worker object may be interrogated by inspecting the object’s $state property. This will return either:

* kWorkerStateCancelled – The worker has been cancelled.
* kWorkerStateClear – The worker is in a pre-initialised state.
* kWorkerStateComplete – The worker has completed.
* kWorkerStateInit – The worker has been initialised.
* kWorkerStateRunning – The worker is currently running.

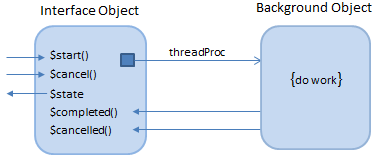
For example:

###### If iWorkerObj.$state=kWorkerStateRunning & iWorkerObj.$waitforcomplete=kTrue   Calculate iMesg as 'Still running (waiting for completion)'   Quit method End If

### How SQL Worker Objects work

A worker object may be thought of as two sub-objects:

1. **Interface Object**  
   This takes the form of a standard Omnis non-visual object and provides the methods and properties described above
2. **Background Object**  
   Normally created/executed on a separate thread, the Background Object performs the actual work of the worker object, calling back to the Interface Object upon completion.



##### Behind each Worker Object, there is hidden Background Object.

#### Detaching Worker Processes

When instructed to $start(), the Background Object completes its work before calling back to the Interface Object’s $completed() or $cancelled() method. For $run(), $completed() is *always* called since the worker object blocks- preventing cancellation.

##### A detached Background Object. The Interface Object may have gone out-of-scope or may now point to a new Background Object.

For $start() however, the Interface Object may legitimately go out-of-scope or otherwise get destructed before the background thread completes. In this situation, the background thread has no object and hence no $completed()/$cancelled() method to call back to. Any results or error information will therefore be discarded.

#### Discarding Running Processes

In the case where the Interface Object remains in scope, it is possible to call $init() and $start() whilst the worker object is still running a previous task. In this case, the $waitforcomplete property determines whether the running process should be allowed to run to completion and call back to the Interface Object to signal completion.

If $waitforcomplete is kFalse, the running process is detached from the Interface Object as if the Interface Object were about to go out-of-scope. In this case however, a new Background Object is created which is then initialised and used to execute the new worker process, and potentially call back to the Interface Object when complete.

If $waitforcomplete is kTrue, Worker Main returns an error to the Interface Object if an attempt is made to re-use the worker object while the Background Object is still running. In this case, the worker object cannot be re-used until $completed()/$cancelled() has been called and the $state changes to indicate completion.

A worker object with its $waitforcomplete property set to kFalse, effectively becomes a “fire and forget” worker object, for example allowing a succession of INSERT, UPDATE or DELETE statements to be continuously dispatched to separate threads using the same worker object.

#### Cancelling Detached Processes

By default, orphaned background threads are allowed to run to completion. When a worker process becomes orphaned it may be preferable to issue a cancel request to the worker, especially where it may be processing a SELECT statement- for which the results will not be retrievable once detached from the Interface Object. This is achieved by setting $cancelifrunning to kTrue before the worker object gets re-used or destructed.

If $cancelifrunning is set to kFalse (the default), orphaned worker threads run to completion before discarding their results and exiting.

#### Alternative Completion Model

From Studio 8.0.2 Worker Objects support an alternative completion model. The $completed and $cancelled methods can optionally be sent directly to another instance. This means you do not need to sub-class the worker object, in order to receive its results. We would recommend that you use object references rather than objects for this technique.

##### Use of $callbackinst allows callbacks to be sent to another class instance.

In order to use this new functionality, there is a new property of worker object instances, called $callbackinst. If you do not use this new property, behavior is unchanged from Studio 8.0.1 and earlier.

For example, if iWorker is a SQL Worker Object (an instance variable in a window class), then within the window instance you can execute:

iWorker.$callbackinst.$assign($cinst)

You need to implement $completed and $cancelled in the window class methods. The parameters are as follows:

1. $completed(*row,object*)  
   where *row* is a parameter of type Row, same definition as that passed to $completed in the sub-classed object when not using $callbackinst.  
   *object* is a parameter of type Object reference (when the worker object is an object reference) or Item reference (when the worker object is an object).  object is the worker object for which $completed is being called.
2. $cancelled(*object*)  
   where *object* is the same as for $completed

### Additional Notifications

The SQL Worker Objects supplied with Studio 8.1 and later support an interim ‘$progress’ method to be called whilst the worker is running. If implemented in the $callbackinst;

* $progress(*row*)   
  will be called with a row parameter containing a single column; ‘Progress’ calculated as a percentage of the total number of SQL queries that will be executed.

Where a work-list/query and bindvar combination is supplied, the total number of queries is calculated by adding the number of times each query will be executed. The received parameter value is suitable for direct assignment to a progress bar component, for example:

###### On evClick   …   Do iWorker.$callbackinst.$assign($cinst)   Do iWorker.$init(lParams) Returns #F   Do iWorker.$start() Returns #F

###### 

###### # This code appears in the window instance’s $progress method Do $cwind.$objs.progress.$val.$assign(pRow.Progress)

The ‘worker’ sample component supplied with the [External Component SDK](http://www.omnis.net/products/components/buildyourown.jsp) also demonstrates this functionality.

## SQL Worker Lists

Currently, you can define a list or row variable from a SQL class (query, schema or table class), and associate a SQL session object with the variable in order to perform various SQL operations on the list, e.g. populate the list from the database, insert a row into the database.

This new feature allows you to specify that the SQL list or row will use a SQL Worker Object of the same DAM type as the SQL session object to perform SQL operations asynchronously (or synchronously, if preferred). Because the worker can run asynchronously, there are some differences in the way that you can use a table class from which the list or row is defined, compared to the way you use the table class with a SQL session object, as in previous versions of Studio. To be specific, there is less scope to override SQL methods using the table class because of the need to execute the worker in a separate self-contained thread.

### Using a Worker in a SQL List or Row

#### $useworker and $synchronous

If you want to use a worker object with your SQL list or row, you need to assign a new property, $useworker to kTrue. $useworker must be assigned after assigning $sessionobject, and once you have assigned $useworker, you can no longer assign $sessionobject, or access $statementobject (the latter is destroyed if present when $useworker is assigned). $useworker cannot be assigned to kFalse.

In addition, there is a new property $synchronous: if true, and $useworker is true, the worker object for the schema or table instance executes synchronously in the current thread rather than asynchronously in a separate thread. $synchronous defaults to false (meaning use another thread).

In addition, Omnis does not expose the worker properties $waitforcomplete and $cancelifrunning.

$waitforcomplete will always be kTrue, to make sure the application is notified of the success or failure of an operation, and $cancelifrunning is not relevant - the table will not invoke a new request until the previous request has completed - requests are queued by the table instance while the worker is busy processing a request.

### Selecting & Fetching Data

Non-worker SQL lists and rows can operate in a nice synchronous manner. So $select() can be used to generate a result set, and $fetch() can be called multiple times to retrieve the result set.

SQL Worker based lists and rows cannot run in this simple synchronous manner, because the result set is generated by the worker in a separate thread. Therefore, worker SQL lists and rows have a new method, $selectfetch that performs both the select and the fetch of the data. It has the following definition:

1. **$selectfetch()**$selectfetch([bDistinct=kFalse, iMaxRows=1, bAppend=kTrue, cText,…])

Note that $selectfetch() cannot be used with a row variable defined from a SQL class, so if you want to fetch data using a worker you must define a list from the SQL class.

Note also that you cannot override $selectfetch() in a table class. The parameters are as follows:

1. **bDistinct**  
   Pass this as kTrue to make the worker use a SELECT DISTINCT query rather than SELECT.
2. **iMaxRows**  
   The maximum number of rows to fetch. Must be between 1 and 10000000 inclusive.
3. **bAppend**  
   Pass this as kTrue to append the fetched data to the list, kFalse to replace the list contents with the fetched data.
4. **cText,…**   
   Any further parameters are treated as SQL text and appended to the generated SELECT or SELECT DISTINCT query.

Any errors that are detected before invoking the worker object, result in a call to $sqlerror in the table instance.

After fetching the data, the worker generates a notification to $completed in the table instance.

### Inserts, Updates and Deletes

When using a worker, you cannot override $insert, $update or $delete in a table class.

When you execute these methods via a worker, the table instance copies the current values of the affected row (rows for $update) into the parameter list for the worker, and then starts the worker.

Any errors that are detected before invoking the worker object, result in a call to $sqlerror in the table instance.

On completion, the worker generates a notification to $completed in the table instance.

### Smart List Methods

When using a worker, you cannot override $dowork, $doinserts, $doupdates, $dodeletes, $doinsert, $doupdate or $dodelete. Also, you cannot call $doinsert, $doupdate or $dodelete.

When you call $dowork, $doinserts, $doupdates or $dodeletes, the table instance generates a single query for each of the relevant operations insert, update and delete. The instance then copies bind variable values into a list, for each set of rows to be inserted, updated or deleted. Finally, the table instance starts the worker with the copied data as its parameters. When the worker completes, the worker generates a notification to $completed, that identifies any rows for which an error occurred, with information about the error.

Note that as soon as you call $dowork, $doinserts, $doupdates or $dodeletes, the smart list updates just before starting the worker

Any errors that are detected before invoking the worker object, result in a call to $sqlerror in the table instance.

### Completion Row

The table instance properties $rowsaffected and $rowsfetched are not relevant when using a worker.

$completed in the table instance is passed a row variable parameter with columns as follows:

1. **errorcode**  
   An error code. Zero means the worker was successfully passed the query and bind variables. Note that the query or queries may still have failed - see the errors column.
2. **errortext**  
   Error text describing the errorcode.
3. **work**  
   The list of queries and bind variables that were passed to the worker. This has the usual structure for SQL workers - two columns, query and bindvars.
4. **errors**  
   If errorcode is zero, this is a list of queries that generated a SQL error of some sort. This has the same structure as the Errors column generated by a SQL worker in the worker completion row.
5. **rowsFetched**  
   If a call to $selectfetch successfully fetched some rows, this is the number of rows fetched.