# Chapter 9—Server-Specific Programming

This chapter contains server-specific information for each of the proprietary databases and middleware configurations that Omnis Studio supports, including information for:

* Oracle
* Sybase
* DB2
* MySQL
* PostgreSQL
* SQLite
* ODBC
* Amazon SimpleDB

Every DBMS has its own specific, extra features that are not part of the SQL standard. Some of these features are supported by sending a specific database command to the server using the standard $prepare(), $execute(), and $execdirect() methods. Others are implemented as special session and statement object properties and methods.

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

### Server and Clientware Compatibility

Some aspects of functionality and compatibility are subject to frequent change with new versions of server software or clientware. Check the Omnis Developer website for details of software versions supported and middleware configurations at: [www.omnis.net/developers/resources/dams/](http://www.omnis.net/developers/resources/dams/)

There you can view the latest information about the Clientware supported by the different server databases supported in the current version Omnis Studio.

### 64-bit DAMs

The DAMs provided with the 64-bit version of Omnis Studio 8.0 or higher 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.

## Oracle

This section contains additional information you need to access an Oracle database, including server-specific programming, PL/SQL, data type mapping to and from Oracle, as well as troubleshooting. For general information about logging on and managing your database using the Omnis SQL Browser, refer to the earlier parts of this manual.

### Properties and Methods

In addition to the “base” properties and methods described in the *SQL Programming* chapter, the Oracle DAM provides the following additional features.

#### Session Methods

|  |  |
| --- | --- |
| Method | Description |
| $proxyas() | SessionObj.$proxyas(cUsername [,cPassword, lRoles]). Allows the supplied user to connect to Oracle through the current connection, which must already be logged-on. The session then acquires the roles and privileges associated with that user. An additional list of roles to be used with the proxy session can also be supplied if required. The list should consist of a single column of type Character. The password should be supplied if the proxy requires authentication by password.  $proxyas() can be called repeatedly with different usernames if required, in which case the current proxy is implicitly terminated before the new proxy is established. |
| $endproxy() | SessionObj.$endproxy(). Explicitly tests for and terminates a proxy session if one exists, returning the session roles and privileges back to those of the primary connection. |

#### Session Properties

|  |  |
| --- | --- |
| Property | Description |
| $nullasempty | Default value kFalse. If kTrue Null values are converted to empty values when fetched from the database. |
| $emptyasnull | When kTrue, empty Omnis strings are inserted as NULL. When kFalse, they are inserted as chr(0). $emptyasnull defaults to kTrue. |
| $trailingspaces | Default value kFalse. If kFalse is specified any trailing spaces on character data being inserted are stripped. If kTrue is specified, trailing spaces are kept. |
| $maxvarchar2 | Default is 2000. Specifies the length above which Omnis character columns will be mapped to the LONG/CLOB data type in Oracle 7 & 8. The max value is 4000 for DAMORA8 and 2000 for DAMORA7. Setting $maxvarchar2 to zero forces all character data to be mapped to the LONG/CLOB data type. |
| $nationaltonclob | Oracle 8 or above: is used to alter the default mapping of Omnis Character and National types. By default, Omnis Character and National fields with a subtype greater than $maxvarchar2 are mapped to the NCLOB data type. By setting $nationaltonclob to kTrue only National fields with a subtype >$maxvarchar2 are mapped as NCLOBs. Character fields with subtype >maxvarchar2 are mapped as non-Unicode CLOBs. Character fields mapped in this way are subject to data loss/truncation where such fields contain Unicode characters. |
| $nationaltonvarchar | Only available in the Unicode DAM, $nationaltonvarchar is used to alter the default mapping of Omnis Character and National types. By default, Omnis Character and National fields with a subtype <= $maxvarchar2 are mapped to the NVARCHAR2 data type. By setting $nationaltonvarchar to kTrue only National fields with a subtype <= $maxvarchar2 are mapped as NVARCHAR2. Character fields with subtype <= $maxvarchar2 are mapped as non-Unicode VARCHAR2 columns. Character fields mapped in this way are subject to data loss/truncation where such fields contain Unicode characters. Please note VARCHAR2 and NVARCHAR2 columns are limited to 4000 bytes. Hence NVARCHAR2 columns are limited to 2000 UTF-16 characters. |
| $internalcharmapping | If set to kFalse, conversion of non-Unicode character data to and from the Omnis character set is disabled, even when $charmap=kSessionCharMapOmnis or kSessionCharMapTable, thus allowing custom character maps to be used with *native* characters if required. Default setting is kTrue. |
| $nativewarncode | A warning code issued by the clientware in response to the last session method to be executed. |
| $nativewarntext | A warning message issued by the clientware in response to the last session method to be executed. |
| $booltonum | If set to kTrue, Omnis Boolean fields will be mapped to the Oracle NUMBER(1,0) data type. Bound kTrue and kFalse values will be written as 1 and 0, respectively. $createnames() will return NUMBER(1,0) as opposed to VARCHAR2(3). When set to kFalse (default), the old behavior is retained. |
| $authmode | Oracle 8 or above: Specifies the authentication mode to be used with the connection. By default, kAuthDefault is used. If you have sufficient privileges however, kAuthSysOper or kAuthSysDba can be supplied. $authmode must be set before executing $logon(). |
| $credentials | Specifies the type of credentials to be used for establishing the connection. Valid modes are: -Authentication via username and password (kCredRDBMS) -Authentication using the Windows user account (kCredExt) To establish a proxy connection, please refer to the $proxyas() method. |
| $longchartoclob | If set to kTrue, Omnis large character fields > $maxvarchar2 in length will be mapped to the CLOB data type. This affects inserts and updates as well as the text returned by $createnames(). When set to kFalse, the DAM maps long character fields to the Oracle LONG data type (Oracle7 behaviour). This property is read-only for DAMORA7 and defaults to kTrue for DAMORA8. |
| $binarytoblob | If set to kTrue, Omnis binary fields will be mapped to the BLOB data type. This affects inserts and updates as well as the text returned by $createnames().When set to kFalse, the DAM maps binary fields to the Oracle LONG RAW data type (Oracle7 behaviour). This property is read-only for DAMORA7 and defaults to kTrue for DAMORA8. |
| $newpassword | If set, the DAM will attempt to change the password during $logon(). Intended to allow expired passwords to be changed but can also be used in the general case. If successful, the logon will proceed as normal, the $password property will be updated and $newpassword will be cleared. When changing the password, the existing username and password should be passed via the $logon() method as normal. |
| $truetext & $falsetext | Studio 5.0 and later. Contain the text that will be inserted for Boolean bind variables. Where these values where previously taken from the Omnis localization datafile, these properties now permit localization-independent values to be inserted, e.g. ‘YES’ & ‘NO’. For backwards compatibility, default values are taken from Omnisloc.df1. Affects the text returned by $createnames() and the size of buffers used to insert data. |
| $querytimeout | Studio 4.3.2/5.0.1 and later. This is the timeout in seconds for any statement executed via $execute() or $execdirect(). Designed to detect Unix network hangs, this property has no effect for Win32 and macOS. When a timeout occurs, the connection is marked bad and a re-connect is necessary. Default value is 10 seconds. |

#### Statement Methods

|  |  |
| --- | --- |
| Methods | Description |
| $plsql() | StatementObj.$plsql(cPLSQLtext[,iColNo...]). This function should be used instead of $prepare() when you want to call server procedures that contain bound OUT or IN/OUT parameters. |
| $prepareforupdate() | StatementObj.$prepareforupdate(vTableDef,cTablename[,cWhere]) creates and prepares a 'select for update' statement for specific use with positioned updates and deletes. vTableDef is a row or list variable defined with columns to be selected. |
| $posdelete() | StatementObj.$posdelete(oStatement) deletes a row positioned by the specified statement object. oStatement is a statement object prepared previously using $prepareforupdate(), and executed. |
| $posupdate() | StatementObj.$posupdate(oStatement,wRow) updates a row positioned by the specified statement object. oStatement is a statement object prepared previously using $prepareforupdate(), and executed. |

#### Statement Properties

|  |  |
| --- | --- |
| Property | Description |
| $nativewarncode | A warning code issued by the clientware in response to the last statement method to be executed. |
| $nativewarntext | A warning message issued by the clientware in response to the last statement method to be executed. |
| $plsqlarraysize | When retrieving data into an Omnis list via PlSql, the number of rows that will be fetched is not known until the plsql executes. Historically, the DAM reserved a pre-determined buffer size of 32512 bytes per list column to be fetched. If the actual number of rows fetched \* column size (in bytes) exceeds this limit for a given column, ORA-06513 is returned.  The column buffer size is now set according to $plsqlarraysize (default value 32512), thus the buffer size can be raised (or lowered) as required in order to accommodate the entire result set. |

### Connecting to your Database

To connect to your database, you need to create a session object with a subtype ORACLE8SESS (or the legacy subtype: ORACLE7SESS). In order to log on to the database using the SessObj.$logon() method, the hostname must contain a valid Oracle host alias previously generated by the Oracle client tools.

### Mixing Unicode and Non-Unicode Data Types

This section summarises recent changes made to the Unicode Oracle Object DAM designed to enable insertion and retrieval of mixed ANSI and Unicode character types.

In the case of Oracle 8i and later, these data types are:

|  |  |
| --- | --- |
|  |  |
| CHAR | Fixed single-byte character data, limited to 2000 bytes. |
| NCHAR | Fixed multi-byte character data, limited to 2000 bytes. (1000 UCS-2 encoded characters) |
| VARCHAR2 | Varying length, single-byte character data, limited to 4000 bytes. |
| NVARCHAR2 | Varying length, multi-byte character data, limited to 4000 bytes. (2000 UCS-2 encoded characters) |
| CLOB | Character Large Object- single-byte character data. |
| NCLOB | National Character Large Object- multi-byte character data. |
| LONG | Varying length, single-byte character data, limited to 2GB. Supported for backward compatibility only. |

By default, the Unicode Oracle DAM maps all Omnis character data to the NVARCHAR2 and NCLOB data types, dependent on the field length of the Omnis bind variable. However, the Oracle DAM provides session properties which affect the Omnis to Oracle data type mappings:

1. $nationaltonvarchar   
   If set to kTrue, Character and National data types are treated differently when being inserted to VARCHAR2 / NVARCHAR2 columns.
2. $nationaltonclob   
   If set to kTrue, large Character and National data types are treated differently when being inserted to CLOB / NCLOB columns.
3. $maxvarchar2   
   Sets the byte limit above which Omnis character fields will be mapped to CLOB/NCLOB data types as opposed to VARCHAR2 / NVARCHAR2 columns. The maximum value is 4000 bytes.
4. $longchartoclob   
   If set to kTrue (the default), Omnis large character fields > $maxvarchar2 in byte length will be mapped to the CLOB/NCLOB data type. If set to kFalse, the LONG data type is used.

#### Reading Unicode and Non-Unicode Data

The Oracle DAM automatically detects the data type of retrieved character columns and converts the data accordingly. There is no need to modify any properties in order to retrieve mixed ANSI and/or Unicode Data.

### Insertion/Update of CHAR and VARCHAR2 data

To write short character data to ANSI columns it is necessary to set $nationaltonvarchar to kTrue. In this mode, Omnis Character fields will be mapped to VARCHAR2 and National fields will be mapped to NVARCHAR2.

When set to kFalse (the default), both Character and National types will be mapped to NVARCHAR2.

### Insertion/Update of CLOB data

Where the Omnis field length exceeds $maxvarchar2, the DAM will map to either CLOB, NCLOB or LONG dependent on the value of the $nationaltonclob and $longchartoclob properties. To write long character data to ANSI CLOB columns, it is necessary to set $nationaltonclob to kTrue. In this mode, Omnis Character fields will be mapped to CLOB and National fields will be mapped to NCLOB. When set to kFalse (the default), both Character and National types with byte sizes exceeding $maxvarchar2 will be mapped to NCLOB.

Note that where Omnis fields are mapped to NCLOB columns, $maxvarchar2 is interpreted as the length in bytes. Thus when set to 4000, this mapping will be applied for Character and/or National fields with a field length > 2000 characters

### Fetching Very Large Objects

The Oracle DAM has the ability to fetch very large objects (up to 2GB) directly to the local file system. Two new properties have been added to the session object:

1. $filethreshold   
   the file threshold which is initially set to 50MB
2. $filedirectory   
   the directory to receive the file which is initially set from the USERPROFILE environment variable on Windows or HOME on macOS and Linux

Any CLOB, NCLOB, BLOB or BFILE column which exceeds $filethreshold will now be fetched in chunks using $lobchunksize directly to $filedirectory. The filename used will conform to “colname\_xxxxxx.BIN” where xxxxxx is a unique identifier (based on #CT). Any character data written to file will be converted to UTF8, otherwise raw data will be written. For BFILEs this means that changing the file extension later (e.g. from .BIN to .AVI) will result in a facsimile of the original file. When a VLOB is written to file, its filename is returned into the result list column. Since the result column was previously described as binary it is necessary to extract the filename using the utf8tochar() function, e.g. Calculate filename as utf8tochar(lResult.1.colLOB).

#### Using Worker objects to fetch VLOBs

Fetching VLOBs on the main thread can cause Omnis to pause while the data is being transferred. Therefore, for large transfers it may be preferable to SELECT and FETCH each VLOB using an Oracle worker object. The main thread is then free to continue and will be notified when the VLOB has been fetched.

### Insertion/Update of LONG data

When $longchartoclob is set to kFalse, Omnis Character and National fields which would otherwise map to CLOB or NCLOB will be mapped to the LONG data type. Since Oracle tables may contain only one column of type LONG, this may lead to problems if not used judiciously.

### PL/SQL

Oracle does not support the remote procedure call methods such as $rpc() which are described in the *SQL Programming* chapter. Server procedures can be executed via PL/SQL. The Oracle DAM fully supports Oracle PL/SQL; a procedural language that the server executes.

You create a PL/SQL script and send it to Oracle in a similar way as any SQL statement and the server executes it. The statement object method $plsql() should be used instead of $prepare() when you want to call server procedures that contain bound OUT or IN/OUT parameters. Any PL/SQL bind variables being passed to Oracle should be passed as Omnis bind variables i.e. @[Xvar]. When $plsql() is called with bind variables, the DAM will check to see whether any return values are present after execution and will return them back into the Omnis variable. This does not happen after a $prepare(). If you are creating stored procedures or executing a server procedure for which you do not expect a return value, $prepare() will be sufficient.

###### # lEmpId is local Integer 32-bit with initial value of 0 Do StatObj.$plsql("begin select empno into @[lEmpId] from scott.emp where ename = 'JONES'; end;") Do StatObj.$execute() If lEmpId <>7566   # Incorrect value returned from procedure End If

After the PL/SQL executes, the Omnis variable lEmpId has the value associated with the column id from the row with the name ‘Jones’.

The Oracle DAM supports select tables returned through PL/SQL procedures. However Oracle can only return single column tables (arrays). The statement object method, $plsql(), has optional parameters that follow the cPLSQLtext parameter. These are necessary when calling server procedures that return single column tables. To bind a column of an Omnis list to the select table being returned requires a list variable to be bound in the SQL statement and the list column number to be passed as an additional parameter. If more than one select table is being returned, multiple lists will need to be bound and a column number parameter passed for each list, in the same order as the lists are bound. The bound lists do not have to be different lists, the same list can be bound more than once in the PL/SQL statement, but you must take care to specify a different column number for each occurrence of the bound list. If no column number parameter is passed for a bound list, the first column of the list is used by default.

Consider the following example table and PL/SQL package:

|  |  |  |  |
| --- | --- | --- | --- |
| **Table: accounts** |  |  |  |
| id  NUMBER(3,0) | name  NVARCHAR2(256) | balance  NUMBER(16,2) | limit  NUMBER(16,2) |
| 1 | Bill | 2000 | 2200 |
| 2 | Sally | 120 | 100 |
| 3 | Bob | 1000 | 190 |
| 4 | John | 1700 | 1500 |
| 5 | Graham | 3000 | 21087 |
| 6 | Helen | 2000 | 1860 |
| 7 | Betty | 9000 | 1490 |
| 8 | Walter | 25000 | 17200 |
| 9 | Sarah | 9100 | 10000 |

###### create or replace package test as   type account\_name is table of accounts.name%TYPE index by binary\_integer;   type account\_balance is table of accounts.balance%TYPE index by binary\_integer; end test;

###### create or replace procedure credit(accnum IN number, amount IN number,pname out test.account\_name, pbalance out test.account\_balance) is   cursor c1 is select name,balance from accounts where balance > limit;   row\_count BINARY\_INTEGER; begin   row\_count := 1;   update accounts set balance = balance + amount where id = accnum;   open c1;   LOOP     FETCH c1 INTO pname(row\_count),pbalance(row\_count);     row\_count := row\_count + 1;     exit when c1%NOTFOUND;   end LOOP;   close c1; end;

###### # Local variables: # lName=Character 256, lBalance=Number2dp, lCreaditList=List Do SessObj.$blobsize.$assign(32767) ## Prevents ORA-06505 Do lCreditlist.$define(lName,lBalance) Do StatObj.$plsql('begin credit(2,490,@[lCreditlist], @[lCreditlist]); end;',1,2) Returns #F Do StatObj.$execute() Returns #F If Creditlist.$linecount<>6   # Incorrect data returned from stored procedure End If

In the above example, the same list has been bound twice, the first bind variable binds the first column of the list and the second bind variable binds the second column as defined by the second and third parameters of the $plsql() method.

After the PL/SQL procedure executes, lCreditList should contain 6 rows as follows:

|  |  |
| --- | --- |
| name | balance |
| Sally | 610.00 |
| Bob | 1000.00 |
| John | 1700.00 |
| Helen | 2000.00 |
| Betty | 9000.00 |
| Walter | 25000.00 |

The balance of account id 2 (Sally) has been increased by 490. The procedure returns details of those accounts where the balance column is greater than the limit column.

### Positioned Updates and Deletes

You can use positioned updates and deletes to update and delete specific rows from the select table you are fetching. To enable positioned updates and deletes the statement object method, $prepareforupdate(), should be used. This method creates and prepares a 'select for update' statement for specific use with positioned updates and deletes. A 'select for update' SQL statement can be prepared using the $prepare() method. However, it will not store the current ROWID in the statement object.

You can use $prepareforupdate() in conjunction with $posupdate() and $posdelete() to update or delete a row, which is determined by the current row in the specified statement. The 'select for update' statement is built based on the parameters passed. The columns will be derived from the list or row passed as the first parameter. If the cTablename parameter is omitted, the name of the list/row is assumed to be the name of the table. A SQL WHERE clause is appended to the select statement if it has been specified. After this statement has been executed the last row fetched will be seen to be the current row. If the statement does not perform a fetch, there will not be a current row.

**Note:** For all position update and delete functionality the transaction mode must be kSessionTranManual.

###### # lEmpSt,lEmpUpSt,lEmpDelSt are Statement Object instances derived from the same session # lTableName is a Character(32) variable # iTableList and lTempList are list variables defined from schema class scPos1 # lDataRow is a row defined from schema class scPos1 # Fetch the row Calculate lTableName as $classes.scPos1.$servertablename() Do lEmpSt.$prepareforupdate(iTableList,lTableName) Do lEmpSt.$execute() Do lEmpSt.$fetch(lDataRow,1) # Update this row Calculate lDataRow.vala as 5 ## change the value of one of the columns Do lEmpUpSt.$posupdate(lEmpSt,lDataRow) Do lEmpUpSt.$execute() # Fetch the next row Do lEmpSt.$fetch(lTempList,2) # delete this row Do lEmpDelSt.$posdelete(lEmpSt) Do lEmpDelSt.$execute()

### Oracle 8 Data types

The Oracle8 DAM (DAMORA8) supercedes the older Oracle7 DAM (DAMORA7, which is no longer in development). DAMORA8 has been specifically written to connect to Oracle8 (and later) databases but can be used against an Oracle7 server using the recommended clientware. In this case, the DAM will encounter restrictions as described below, mainly concerned with data type mapping of large objects.

#### Large Objects (LOBS))

CLOBs, NCLOBs and BLOBs are data types introduced for Oracle 8 that deal with large objects. Internal LOBs (BLOBs, CLOBs, NCLOBs) are stored in the database tablespaces in a way that optimizes space and provides efficient access. These LOBs have the full transactional support of the database server. The maximum length of a LOB/FILE is 4 gigabytes. Internal LOBs have copy semantics. Thus, if a LOB in one row is copied to a LOB in another row, the actual LOB value is copied, and a new LOB locator is created for the copied LOB.

The Oracle8 DAM uses locators to point to the data of a LOB or FILE. These locators are invisible as the DAM performs operations on the locator to insert, update, delete and fetch the values. This means that you are only ever dealing with the values of the LOBs and not the locators.

You can work with the locators rather than just the values using PL/SQL in conjunction with the dbms\_lob package. Further information can be found in the Oracle8i Supplied Packages Reference.

External LOBs (FILEs) are large data objects stored in the server's operating system files outside the database tablespace. FILE functionality is read-only. Oracle currently supports only binary files (BFILEs). The Oracle8 DAM uses locators to point to the data of a FILE. The FILE locator will be invisible, as the DAM will return the value of the external file and not the locator when performing transactions with the BFILE data type. Even though the BFILE data type is read-only you can insert a directory alias and filename into the column. These values are assigned to a single Omnis binary variable and separated by the '&' symbol. The DAM will assign these values to the locator so that when a fetch is performed on the locator the binary representation of the external file corresponding to the alias and filename will be returned. An example is shown below.

###### # A Directory alias needs to be created on the server that points # to an OS folder myStat.$execdirect("create or replace directory sound as 'c:\bfiles'") # BFILE1 and BFILE2 are Omnis variables of type Binary. # The variable is calculated as # '<DirectoryAlias>&<Filename>'. Calculate BFILE1 as 'sound&wav2.wav' Calculate BFILE2 as 'sound&wav3.wav' myStat.$execdirect('insert into bfiletest values(1,@[BFILE1],@[BFILE2])') # You can now select the data back and this time you can receive the binary representation of the file. myStat.$execdirect('select \* from bfiletest') myStat.$fetch(myRow) # The values contained in col2 and col3 of myRow can now be # written to the local drive using the Omnis Fileops commands. Calculate File as 'c:\windows\desktop\wavtest.wav' Do Fileops.$createfile(File) Returns lErr Do Fileops.$openfile(File) Returns lErr Do Fileops.$writefile(myRow.2) Returns lErr Do Fileops.$closefile() Returns lErr

#### Ref Cursor Data Types

The REF CURSOR is an Oracle 8 data type that is used to point to a set of results from a multi-row query. When executing a multi-row query, Oracle opens an unnamed work area that stores processing information. To access the information, you can use a variable of type REF CURSOR, which points to the work area. To create cursor variables, you define a REF CURSOR type and then declare cursor variables of that type.

A REF CURSOR type can be defined in an Oracle Package object. For example:

###### create or replace package OmnisPackage as type cursorType is ref cursor; end;

This data type can be used in other Oracle objects, such as procedures and functions in order to process result sets. An example of a Stored function follows:

###### create or replace function OmnisFunction return OmnisPackage.cursorType as l\_cursor OmnisPackage.cursorType; begin open l\_cursor for select \* from scott.dept; return l\_cursor; end;

An example of a stored procedure that uses the defined REF CURSOR type follows:

###### create or replace procedure OmnisProcedure ( p\_cursor in out OmnisPackage.cursorType ) as begin open p\_cursor for select ename, empno from scott.emp order by ename; end;

Cursor variables are like pointers, which hold the memory location (address) of some item instead of the item itself. So, declaring a cursor variable creates a pointer, not an item.

REF CURSOR data types can be returned in three different ways; via a PL/SQL block, an Oracle Stored Function or an Oracle Stored Procedure. The REF CURSOR type is a pointer to a result set. The SQL statement that returns the REF CURSOR must be prepared using the $plsql() method. The Oracle 8 DAM maps the REF CURSOR type to an Omnis Oracle8 Statement Object\*. The statement object will be created by the DAM and will belong to the same session object as the Statement object that prepared the initial SQL. It will have a $state of kStatementStateExecuted and, assuming that there is data in the result set, will have $resultspending set to kTrue. Therefore, the statement object will be in a ‘Ready-For Fetch’ state. Below are examples of the three ways to return and use a REF CURSOR in Omnis. The connection code and the creation of initial Statement Object (myStatement) have been removed for clarity.

\*As of Studio 5.2, the Object variable used to return the REF CURSOR result set may instead be passed as an Object reference if preferred.

#### PL/SQL Block

The PL/SQL method does not require any SQL objects created on the server. All the PL/SQL code can be encapsulated in an Omnis Statement block.

###### # declare vars: cursor1 (Object), myList1 (List) Begin statement Sta: begin Sta: OPEN @[cursor1] FOR SELECT \* FROM scott.emp; Sta: end; End statement If myStatement.$plsql()   If myStatement.$execute()     Do cursor1.$fetch(myList1,kFetchAll)     # myList1 will contain the rows of the result set.   Else     OK message Error {[ myStatement.$nativeerrortext]}   End If Else   OK message Error {[ myStatement.$nativeerrortext]} End If

#### Stored Functions

Returning a REF CURSOR from a Stored Function requires an Oracle Stored Function on the database. The Function must have a return type that has been defined as a REF CURSOR. For this example we will assume that the example Oracle Stored Function described above has been created on the server.

###### # declare vars: cursor2 (Object), myList2 (List) If myStatement.$plsql('begin @[cursor2] := OmnisFunction; end;')   If myStatement.$execute()     Do cursor2.$fetch(myList2,kFetchAll)     # myList2 will contain the rows of the result set.   Else     OK message Error {[ myStatement.$nativeerrortext]}   End If Else   OK message Error {[ myStatement.$nativeerrortext]} End If

#### Stored Procedures

Returning a REF CURSOR from an OUT or IN OUT parameter of a Stored Procedure requires an Oracle Stored Procedure on the database. The Procedure must have an OUT or IN OUT parameter type that has been defined as a REF CURSOR. For this example we will assume that the example Oracle Stored Procedure described above has been created on the server.

###### # declare vars: cursor3 (Object), MyList3 (List) If myStatement.$plsql('begin getemps(@[cursor3]); end;')   If myStatement.$execute()     Do cursor3.$fetch(myList3,kFetchAll)     # myList3 will contain the rows of the result set.   Else     OK message Error {[ myStatement.$nativeerrortext]}   End If Else   OK message Error {[ myStatement.$nativeerrortext]} End If

### Oracle 9i Data types

The Oracle 8 Object DAM includes support for data types added for Oracle 9i, namely the XML and URI data types. The XML data type lets you store native XML documents directly in an Oracle database and eliminates the need to parse the documents coming into and out of the database. Server specific properties and methods have been added to the DAM to support these enhanced database operations.

Oracle 9i also introduced several new Universal Resource Identifier (URI) types. These are used to identify resources such as Web content anywhere on the Web and can be used to point to data either internally or externally from the database itself. In addition to support for URIs, the Oracle DAM includes support for querying and other abstract functions provided for the URI types.

These changes were introduced with Omnis Studio version 3.2. The old-style, single threaded DAM (DORACLE8) can connect to Oracle 9i databases, but does not support the XML and URI data types.

#### XMLType

The XMLType is a system defined data type with predefined member functions to access XML data. You can perform the following tasks with XMLType:

1. Create columns of XMLType and use XMLType member functions on instances of the type.
2. Create PL/SQL functions and procedures, with XMLType as argument and return parameters.
3. Store, index, and manipulate XML data in XMLType columns.

#### URIType

The URIType is an abstract object type that can store instances of HttpUriType or DBUriType**.** Universal Resource Indicator references can point to XML, HTML and custom internet content which can be located either locally within the database, externally to the database but local to the server or remotely across an internet or network connection.

#### DBUriType

The DBUriType can obtain data pointed to by a DataBaseUri-reference. A DBUri-Ref is a database relative URI that is a special case of the Uri-ref mechanism, where ref is guaranteed to work inside the context of a database and session. This ref is not a global ref like the HTTP URL, instead it is local ref (URL) within the database.

#### HttpUriType

The HttpUriType implements the HTTP protocol for accessing remote pages.

#### UriFactoryType

It is not possible to generate table columns using the UriFactoryType. Rather, this is a PL/SQL package containing factory methods that can be used to generate the appropriate instance of the Uri types without having to hard code the implementation in the program. Custom URI types can be defined and registered using this package.

For further information on the application of these data types, refer to the *Oracle9i Application Developer's Guide – XML*.

### Retrieving XML and URI data

In Oracle9i version 9.1, it is not possible to directly SELECT data from columns defined using these types. Instead, the appropriate accessor functions should be used.

XMLType provides the extract(), getClobVal(), getStringVal(), and getNumberVal() functions for data query and retrieval. The extract() function has to be used in conjunction with one of the data type conversion functions, since it returns an object of type XMLType. The following example SQL statement can be used to extract an XML document from an XMLType column:

###### SELECT a.xmlcol.extract(‘\*’).getStringVal() AS mycol FROM mytable a

If required, the XPath expression parameter to the extract() function can be supplied using a character bind variable. For further information on the extract() function and the supported Xpath syntax, refer to the *Oracle9i Application Developer's Guide – XML*.

URITYPE and its derivatives provide the getClob(), getUrl() and getExternalUrl() functions for data retrieval. When getClob() is executed, the URL stored in the database column is read. The document pointed to by the URL is then accessed and returned as CLOB data which can be read into Omnis:

###### SELECT a.myuri.getclob() AS mycol FROM mytable a

The getUrl() and getExternalUrl() functions return the URL contained in the database column. (getExternalUrl() differs from getUrl in that it *escapes* the URL so that it better conforms to the URL specification):

###### SELECT a.myuri.geturl() AS myurl FROM uritest a

The getClob(), getUrl() and getExternalUrl() functions can be overridden when creating custom URI types as defined using the UriFactoryType. For information on the UriFactoryType, refer to the *Oracle9i Application Developer's Guide – XML*.

### Inserting XML and URI data

To insert XML data into an XMLType column, the data must be a valid XML document or fragment. This is because XMLType validates the XML before storing it. A simple insert statement would be of the form:

###### INSERT INTO xmltable VALUES(…,sys.XMLType.createXML (‘<DOC> </DOC>’),…)

If required, the XML text can be supplied using a character bind variable.

XML data can also be supplied as CLOB data by inlining a SELECT statement (or some other expression which returns a CLOB):

###### INSERT INTO xmltable SELECT id, sys.XMLType.createXML(myclob) FROM clobtable;

Inlining ensures that createXML() receives a CLOB field, which is not possible from Omnis since CLOBs are converted into Omnis character strings when fetched.

To insert a URL into a URIType column or one of its derivatives, use the createUri() function. For example:

###### INSERT INTO uritest VALUES (…,sys.httpUriType.createUri(‘http://www.omnis.net’),…)

If required, the URL can be supplied using an Omnis character bind variable.

### Updating XML and URI data

In version 9.1, Oracle9i stores XMLType internally using the CLOB data type. Updates on CLOBs have to be performed on the entire column and for this reason, when updating an XMLType column, it is necessary to re-insert the XML data.

###### UPDATE xmltest SET xmlcol = sys.XMLType.createXml(@[iXMLText]) WHERE IDcol = @[iRecordNum] Similarly, with URITypes, updates are performed as follows: UPDATE uritable SET uricol = sys.httpUriType.createUri(@[iHTMLURL]) WHERE IDcol = @[iRecordNum]

### Oracle Data Type Mapping

The following tables describe the data type mapping for Omnis and Oracle.  
Mappings are for the Oracle8 object DAM and Oracle versions 8i – 11g and later.

#### Omnis to Oracle

|  |  |
| --- | --- |
| Omnis Data Type | ORACLE8 Data Type |
| **CHARACTER** |  |
| [1]Character/National <= the value of  $maxvarchar2 (default is 2000) | NVARCHAR2(n) |
| [1]Character/National > the value of  $maxvarchar2 (default is 2000) | NCLOB |
| **DATE/TIME** |  |
| Short date (all subtypes) | DATE |
| Short time | DATE |
| Date time (#FDT) | DATE |
| **NUMBER** |  |
| Short integer (0 to 255) | NUMBER(3, 0) |
| Integer 64 bit | NUMBER(19,0) |
| Integer 32 bit | NUMBER(11, 0) |
| Short number 0dp | NUMBER(10, 0) |
| Short number 2dp | NUMBER(10, 2) |
| Number floating dp | FLOAT |
| Number 0..14dp | NUMBER(16, 0..14) |
| **OTHER** |  |
| Boolean | [2]VARCHAR2(3) |
| Sequence | NUMBER(11, 0) |
| Picture | [3]BLOB |
| Binary | [3]BLOB |
| List | [3]BLOB |
| Row | [3]BLOB |
| Object | [3]BLOB |
| Item reference | [3]BLOB |

##### [1] Dependant on the values of $nationaltonvarchar, $nationaltonclob and $maxvarchar2 [2] Dependant on the value of $booltonum [3] Dependant on the value of $binarytoblob

#### Oracle to Omnis

|  |  |
| --- | --- |
| Omnis Data Type | ORACLE8 Data Type |
| **CHARACTER / BINARY** |  |
| CHAR / NCHAR | Character |
| VARCHAR2 / NVARCHAR2 | Character |
| CLOB / NCLOB | Character |
| BLOB | Binary |
| BFILE | Binary |
| LONG | Character |
| RAW | Binary |
| LONG RAW | Binary |
| **DATE/TIME** |  |
| DATE | Date time (#FDT) |
| **NUMBER** |  |
| NUMBER(p,0) p>10 | Integer 64 bit[1] |
| NUMBER(p,s) p<=10 or s>0 | Number floating dp |
| NUMBER ( NUMBER(0,0) ) | Number floating dp |
| FLOAT | Number floating dp |

##### [1] $fetch64bitints must be kTrue

### Oracle Troubleshooting

The following points may help in resolving issues in programming Omnis applications that use an Oracle database.

* When performing transactions that use the new LOB data types the transaction mode must be set to either kSessionTranAutomatic or kSessionTranManual. This is because LOB and file locators cannot be used across transactions. The DAM performs functionality on these locators and when the transaction mode is either automatic or manual, the DAM can control when to commit the command, which would be after all the LOB functionality has been performed. When the transaction mode is server, Oracle commits (or rollbacks) after every statement and any LOB functionality performed by the DAM would result in an error.
* Oracle has a client character set and a server character set. If the two are not the same character set, Oracle will convert between the two. If a character in the client character set also exists in the server character set, Oracle will store that character on the server. If the character doesn’t exist in the server character set, Oracle will do one of two things. Firstly, Oracle will see if there is a close-fit character. For example, if you are trying to insert the character ‘à’, but that particular character is not in the server character set, Oracle will store that character as an ‘a’ as it is a close-fit. If there is not a close-fit character, Oracle will store that character as an ‘unknown’ character on the server. This ‘unknown’ character is usually ‘¿’. If the client character set is the same as the server character set, no conversion takes place and all ASCII values of characters are preserved. On retrieval of characters Oracle will again convert but this time from the characters stored in the Oracle database to the client’s character set being used to retrieve the text.
* As with inserting Oracle will convert a character in the server character set to the same character in the client character set if it exists. If not, it will try a ‘best fit’ solution before returning its ‘unknown’ character. To guarantee that Oracle doesn’t convert the data in the database, the client character set should be the same as the server character set. It is possible to use Omnis’ character maps to change some of the characters which aren’t correct to ones that are preferred (Non-Unicode session objects only). This is useful when Oracle returns a ‘best fit’ character and not the original character.

Further troubleshooting notes, “how-tos” and tips can be found on the Omnis website at: [www.omnis.net/technotes](http://www.omnis.net/technotes)

## Sybase

This section contains the additional information you need to access a Sybase database, including server-specific programming, data type mapping to and from Sybase, as well as troubleshooting. For general information about logging on and managing your database using the Omnis SQL Browser, refer to the earlier parts of this manual.

### Properties and Methods

In addition to the “base” properties and methods documented in the *SQL Programming* chapter, the Sybase DAM provides the following additional features.

#### Session Properties

|  |  |
| --- | --- |
| Property | Description |
| $programname | The program name that is registered by Sybase at logon. The default is $clib().$name. |
| $logontimeout | The timeout in seconds for a logon. The default is 60 seconds. Set this to 0 for no timeout. Note that a timeout is ignored if $failtimeout is kFalse. |
| $querytimeout | Timeout in seconds for a query. The default is 0 for no timeout. Note that a timeout is ignored if $failtimeout is kFalse. |
| $failtimeout | Set to kTrue to raise an error if a timeout occurs. If kTrue and a timeout occurs the connection is marked as dead and the session is logged off. |
| $encryptpassword | Set to kTrue to use password encryption when logging on. The default is kFalse. |
| $cterrorlayer | Layer at which the current session client error occurred. Read only. |
| $cterrororigin | Origin of current session client error. Read only. |
| $cterrorseverity | Severity of current session client error. Read only. |
| $cterrornumber | Error number of current session client error. Read only. |
| $moneydps | This property determines the number of decimal places used to store and display data retrieved from MONEY columns. It is also used when creating schemas- provided that this property is set before dragging the table into the library. $moneydps defaults to 4 for backward compatibility but can be set to 0, 1, 2, 3, 4, 5, 6, 8, 10, 12 or 14. |
| $locale | The locale name that will be used by the connection. This is initially set to the default locale contained in the Sybase locales.dat file. $locale may be set to a different value provided that the DAM is not logged on.  Valid locale strings include locale names or language-character set pairs contained in the locales.dat file for which the corresponding language modules are installed. Assignment fails if the locale information specified cannot be found or is not installed. |
| $nativewarntext | Information or warning text generated by the last operation (read-only) |
| $nationaltounichar | Studio 5.1. When this property is set to kTrue, Omnis National fields will be mapped to Sybase NVARCHAR and Character fields will be mapped to VARCHAR. Also affects the text returned by $createnames(). When kFalse (the default), all Character fields are mapped to VARCHAR columns (supporting the UTF-8 encoding). |
| $sdbsocket | Studio 8.0.2. Server-side TCP socket being used by a data bridge connection (read-only) |

#### Session Methods

|  |  |
| --- | --- |
| Method | Description |
| $setremotepassword() | SessObj.$setremotepassword(cServerName, cPassword). Set a password for a remote server connection. This will fail if the session is logged on. If cServerName is NULL, the password is used as a universal password for all servers with no specified password. |
| $clearremotepasswords() | SessObj.$clearremotepasswords(). Clear all passwords for remote server connections. |

#### Statement Properties

|  |  |
| --- | --- |
| Property | Description |
| $cterrorlayer | Layer at which the current session client error occurred. Read only. |
| $cterrororigin | Origin of current session client error. Read only. |
| $cterrorseverity | Severity of current session client error. Read only. |
| $cterrornumber | Error number of current session client error. Read only. |
| $rpcparamspending | If kTrue this denotes that an rpc parameter result set is pending. |
| $bindshort0dpassmallint | If kTrue, Short number 0dp parameters are bound to the server as SMALLINTs. Otherwise the default mapping is used- NUMERIC(9,0). |
| $emptystringisblank | When set to kTrue, the DAM inserts empty character strings into VARCHAR columns as single space characters. When set to kFalse, a NULL or chr(0) is inserted. $emptystringisblank defaults to kTrue. $emptystringisblank does not affect Omnis character strings >255 characters which map to TEXT columns. Empty TEXT values are always inserted as single space characters. |

#### Statement Methods

|  |  |
| --- | --- |
| Method | Description |
| $cancelresultset() | StatementObj.$cancelresultset(). Cancel the current result set. This will allow any further result sets to be processed. If the statement is using a cursor, the cursor is closed and its results are discarded. |
| $writelob() | StatementObj.$writelob(vVariable, cTablename, cColumnname [,cWHERE-clause, bUselog = kTrue]).  Updates a single image or text column with the value of vVariable. The cTablename, cColumnname and optional WHERE clause identify the table column to be updated. If bUselog is kTrue, changes may be rolled back |

### Connecting to your Database

To connect to your database, you need to create a session object with a subtype of SYBASESESS. In order to log on to the database using the SessObj.$logon() method, the hostname must contain a valid Sybase host alias previously generated by the Sybase client tools (see your Sybase documentation for more details).

### Multiple Select Tables

The Sybase DAM lets you run more than one SQL SELECT statement at once and return multiple select tables, one after the other. If a $fetch() call processes the last row of the current result set, the call returns kFetchFinished. If the statement has further result sets available, the statement’s $resultspending property will be kTrue. The application can then process the next result set using $fetch().

One way to do multiple SELECTs is to code a procedure containing several SELECT statements that is to execute on the server. The example below executes the following procedure (assuming the number of lines in your list is adequate to contain all the rows returned):

###### create proc multi\_select as SELECT firstName, lastName FROM Agents SELECT id, name FROM Customers

To execute this from an Omnis method and to fetch the results:

###### Do tStatement.$execdirect('exec multi\_select') Returns #F Do My\_List1.$define() Do tStatement.$fetch(My\_List1,kFetchAll) Returns #1 If #1=kFetchFinished & tStatement.$resultspending=kTrue   Do My\_List1.$define()   Do tStatement.$fetch(My\_List1,kFetchAll) Returns #1 End If

The Sybase DAM reports the select tables exactly as Open Client reports them. If a select result set has no rows, $fetch() will return kFetchFinished the first time it is invoked for that set.

### Program Name

The Sybase sysprocesses table (in the master database) has a program\_name column that stores a separate name for each connection to the server. The session $programname property lets you put a name into this column for the current session so that you can use it to distinguish multiple sessions.

The default name is the name of the current library, i.e. $clib().$name If you wish to change this you must set this property before logging on to the server, because the value gets set at logon. If the value is set after logon it does not take effect until the session is logged on again. The value persists across logons and logoffs, and $clear() does not reset it.

If the string assigned to the property is too long, the DAM truncates it without reporting an error. The DAM can store a maximum of 255 characters but the program\_name field in the sysprocesses table currently only allows 16.

### Error Handling

If an error is raised on either a Sybase session or statement object, the $errorcode and $errortext properties associated with the object will provide the generic error code and error text. If there is an associated native Sybase server or client error this will be returned in the $nativeerrorcode and $nativeerrortext properties associated with the object.

It is possible for an Omnis command to generate multiple Sybase server and client error messages. If this is the case, the object’s $nativeerrorpending property will be set to kTrue. To retrieve the next set of error information the application can use the $nextnativeerror() method. The DAM will return server errors and then client errors in the order in which they were generated. Any informational messages returned from Sybase are ignored.

If a new Sybase DAM command is issued or a Sybase property is set, the current error information for that object is cleared. The error set for the session is shared between the session and all statements in that session. If a session or statement clears the current error set, any other statement with multiple errors pending will only be able to retrieve the last cached error since the error set will have been cleared.

You should be aware that the Sybase server and client errors reported may have codes and messages that sometimes differ between the macOS and Windows Sybase clients.

The Sybase DAM defines several of its own internal error codes. These are returned in the $nativeerrorcode and $nativeerrortext properties of the session and statement.

|  |  |
| --- | --- |
| Error Code | Description |
| 20000 | The Omnis bind variable could not be mapped to an equivalent Sybase type. |
| 20005 | The session must not be logged on. |
| 20010 | The login timeout value must be >= 0. |
| 20011 | The query timeout value must be >= 0. |
| 20030 | Sybase TEXT and IMAGE columns cannot be bound as part of an RPC call. |
| 20050 | The Omnis field cannot be null or empty. |
| 20051 | The table name must be a character string which is not null or empty |
| 20052 | The column name must be a character string which is not null or empty |
| 20053 | The WHERE-clause must be a character string |
| 20054 | The column to be updated was not a TEXT or IMAGE column |
| 20055 | A memory allocation error occurred during the $writelob command |

If a Sybase session or statement object generates an Open Client error, the error code is decoded into the $cterrorlayer, $cterrorseverity, $cterrororigin and $cterrornumber. Sybase Open Client errors use these 4 error components to provide more detail about the error raised.

If a timeout error occurs and the session’s $failtimeout property is kTrue, a timeout error will be raised, the connection will be marked as dead and the session will be logged off. If the $failtimeout is set to kFalse (the default) the connection or query will be re-tried.

### Large Objects (LOBs)

The Sybase Object DAM can send and retrieve text and image fields which are referred to as LOBs or “large objects”. You can insert, update and fetch the fields using the standard SQL object methods or you can use the $writelob() method to update a text or image field on the DBMS, with the implicit functionality to retrieve a large text or image field.

Transferring BLOBs is very memory-intensive, since each layer of software has a copy of at least part of the blob. Thus, sending a simple 40K picture can demand several times that amount of RAM before it gets passed over to the DBMS. Therefore an application must have sufficient memory resources to transfer large text and image data. The built in chunking mechanism can be used to reduce the amount of memory Omnis requires to transfer a LOB value. For example, the default settings for the session’s $lobthreshold and $chunksize properties ensure that data greater than 32K is sent in chunks no greater than 32K. The chunksize and threshold can be altered to suit the resources available. If a system has more memory, the threshold and chunksize can be increased to send fewer, larger chunks.

There are no limitations, aside from memory concerns, on sending or retrieving multiple LOBs in one SQL statement.

You should not forget to set the textsize parameter on the server. This parameter tells the server to truncate all outgoing values to this setting (see your Sybase documentation for more details). Therefore, if you set the textsize parameter to the default of 32,767 and select a 500K image, you get a 32,767 byte value in Omnis.

###### Do tStatement.$execdirect('set textsize 123456') # this sets the textsize parameter, for this session, to about 123K

This setting is for fetching values only. When fetching LOBs under Mac Classic using the standard commands, you should not set this parameter to its largest value. Increasing this also causes Open Client to allocate more memory to deal with the larger LOBs. Therefore, setting it too small will truncate your fetched data while setting it too large may cause Open Client to kill your connection. If you are retrieving a variety of LOBs, you should try to set it as closely as you can to the size of the largest LOB; you can set this for each SQL statement sent. You can also use the Sybase datalength() function to find out how long the value is that you want to retrieve, and use this to set the textsize parameter.

The Sybase DAM provides a faster and more memory efficient way to update a text or image column through the use of the statement method $writelob(). To use the $writelob() method you must already have the row in the database and the column value that is being updated must have non-NULL data in it.. You would usually create the row with a blank (' ') in the column, for instance, use the $writelob() method to update the value with the LOB data.

###### Do tStatement.$execdirect ("insert into mytable (x, mycol) values (2, ' ')") Do tStatement.$writelob(LOB\_DATA,'mytable','mycol','where x=2',kTrue)

This command places the value of the Omnis field LOB\_DATA into the column mycol of the table mytable in the row where x has the value of 2. Thus, the method places a single LOB value into a location that you specify.

The method is defined as:

###### StatementObj.$writelob(vVariable, cTablename, cColumnname [,cWHERE-clause, bUselog = kTrue])

The **vVariable** parameter is the Omnis variable containing the data to be sent to the server, in the example this is LOB\_DATA. This variable cannot be NULL or empty.

The **cTablename** and **cColumnname** parameters identify the table and column to update. These cannot be NULL or empty.

The **cWHERE-clause** parameter supplies an optional WHERE clause for a SQL SELECT statement, including the word ‘WHERE’. If your WHERE clause is ambiguous Omnis updates the first LOB value it finds, so the value updated may not be the one you intended. Make sure your clause specifies a unique row.

The **bUseLog** parameter denotes whether to log this action in the transaction log. If you do not log the action, you cannot roll it back. The default is kTrue to log the update. Setting this parameter to kFalse requires that the select into/bulkcopy option be set to true with the system procedure sp\_dboption for the database on the DBMS. If you do not wish to log updates, you must consult your documentation and system administrator as this may have significant ramifications on being able to backup and recover your database.

The $writelob() method sets the flag false and sets error information in the same way as a standard statement method.

Sybase recommend that data should be updated using the $writelob() method if the data size exceeds 100K.

### Remote Procedure Calls

The Sybase DAM supports the use of the session method $rpcdefine() to define a remote procedure and the statement method $rpc() to call a procedure. The DAM does not support the statement methods $rpcprocedures() and $rpcparameters().

An application must generate the parameter list passed to $rpcdefine() to describe the parameters in the Sybase procedure. When a $rpc() call invokes the procedure the parameters passed are mapped from their Omnis type definition to the equivalent Sybase type in the same way as standard Omnis bind variables. A $rpc() call will fail if a parameter definition includes an Omnis character or binary field larger than 255 since the parameter will map to a text or image field which are not valid for use as parameters in Sybase stored procedures.

If an rpc definition defines parameters of type kParameterInputOutput, these are treated as output parameters since Sybase does not support updateable input parameters. The DAM cannot update output parameters directly. If the procedure uses output parameters these must still be specified in the call to $rpc() and are returned as a parameter result set which is available when the statement property $rpcparamspending is kTrue. They must be processed by the application in the same way as a normal result set. Any result sets generated by the stored procedure must be processed before the parameter results become available. The return status should not be included in the call to $rpc() as this will be set in the statement $rpcreturnvalue property which is set after the stored procedure results are processed.

The following creates a Sybase stored procedure which takes 2 input parameters and 1 output parameter. This procedure returns two result sets.

###### Do tStatement.$execdirect('create procedure Test\_SYBRPC @parm1 varchar(30), @parm2 varchar(30), @parm3 varchar(60) OUTPUT AS SELECT @parm3 = @parm1+@parm2 SELECT \* from sysusers execute sp\_who RETURN 12345')

A list is used to define this procedure in the Sybase session.

###### Do #L1.$define(#1,#2,#3,#4) Returns #F Do #L1.$add(kInteger,kLongint,0,kParameterReturnValue) Do #L1.$add(kCharacter,kSimplechar,30,kParameterInput) Do #L1.$add(kCharacter,kSimplechar,30,kParameterInput) Do #L1.$add(kCharacter,kSimplechar,30,kParameterOutput) Do tSession.$rpcdefine('Test\_SYBRPC',#L1) Returns #F

The procedure can then be called.

###### Calculate Parm1 as 'Hello ' ## Character 30 Calculate Parm2 as ' There' ## Character 30 Calculate Parm3 as ## Character 60 Do tStatement.$rpc('Test\_SYBRPC',Parm1,Parm2,Parm3) Returns #F

Since the stored procedure generates two result sets these must be processed first.

This will fetch the results from the sysusers and sp\_who queries.

###### If tStatement.$resultspending=kTrue   Do #L1.$define()   Do tStatement.$fetch(#L1,kFetchAll) Returns #1 End If If tStatement.$resultspending=kTrue   Do #L1.$define()   Do tStatement.$fetch(#L1,kFetchAll) Returns #1 End If

The stored procedure return status is placed in the statement’s $rpcreturnvalue property and the parameter result set is then available.

###### Calculate #1 as tStatement.$rpcreturnvalue ## will set #1 to 12345 If tStatement.$rpcparamspending=kTrue   Do #L1.$define()   Do tStatement.$fetch(#L1) Returns #1 End If Calculate Parm3 as #L1.1.1 ## will set Parm3 to 'Hello There'

An RPC or stored procedure can invoke procedures on another server. To do this, the server that the first procedure is running on must log onto the other server and for this you can set up a password to use when logging onto that server. The Sybase session object provides the $setremotepassword() method to allow you to set up remote passwords. This associates the specified password with the specified server for this connection to the DBMS. You can call this multiple times to establish passwords for different servers.

If the server name is NULL, the password is a 'universal' password that you can use with any server for which you haven't already established a password.

To clear all remote passwords use the $clearremotepasswords() session method.

### Multiple Cursors

If a statement is issued without using a cursor, i.e. $usecursor is set to kFalse, any results generated will block the connection and no other operation on any other statement can be performed until the blocking result set is completely fetched or cancelled. To avoid blocking the connection with pending results, use a statement which has the $usecursor set to kTrue. Note that a statement using a Sybase cursor must have a unique statement name and only allows SQL SELECT and EXECUTE procedure commands to be issued.

### Meta-Data Queries

As of Studio 5.1.1, the $indexes() meta data method returns additional information via the DamInfoRow column. The DamInfoRow will be defined with the following columns (as returned by the *sp\_statistics* stored procedure):

|  |  |
| --- | --- |
|  |  |
| TableName | Character column containing the table name passed previously. |
| IndexQualifier | Character column indicating the index owner. For Sybase, this is usually the same as *TableName.* |
| IndexType | Character column indicating the index type,  e.g. “Clustered” or “Non-Clustered”. |
| Collation | Character column indicating the collation type,  either “Ascending” or “Descending”. |
| Cardinality | Integer column containing the number of indexed or unique rows. |
| Pages | Integer column containing the number of pages used to store the index. |

### Logon Problems using the SYBASEDAM

In the event of connection problems, there are a number of Technotes available on the Omnis website which discuss Sybase connection issues in greater detail.

Possible causes:

1. The $SYBASE/interfaces file is missing or the contents are invalid- follow the installation tasks outlined above.
2. The logon hostname does not match the name contained in the $SYBASE/interfaces file- check the contents of the interfaces file, paying attention to upper and lower case characters.
3. The supplied username and/or password were incorrect- check at the server.
4. The xcomp:ini:sybasedam.ini file was not found or one or more of the environment variables are set to incorrect values. Review this file.
5. The DAM does not load. The dynamic linker may be unable to locate the required Sybase client libraries. Check environment variables (DYLD\_LIBRARY\_PATH for macOS, LD\_LIBRARY\_PATH for Linux)

### Sybase Troubleshooting

The following points may help in resolving issues in programming Omnis applications that use a Sybase database.

1. Sybase is a case-sensitive RDBMS. Check the case of the table or column names if you can see a table but cannot select anything out of it
2. Sybase defaults to NOT NULL columns; you must initialize columns to a specific value while inserting data, or insertion will fail
3. Any number with no digits after the decimal point, that is > +/- 231 will generate an error and not be inserted. This is because Sybase tries to parse numbers without decimal points as integers
4. Sybase does not support binding a NULL Boolean field in Omnis to a Sybase bit field
5. Sybase does some character mapping where required, but you may need to do character conversion explicitly using the Omnis character mapping tables.
6. Sybase interprets empty strings as single spaces.
7. Fetching pictures from Sybase stored there by other applications, even in standard formats, is likely to cause problems, since Omnis stores all pictures in a special format. This occurs even in platform-specific graphics formats such as PICT or BMP.
8. The $tables() session method can only report information about tables in the current database and does not return system tables.
9. The $columns() session method can only report information about tables owned by the current user in the current database.
10. The $indexes() session method can only report information about indexes on tables in the current database.
11. Sybase does not allow DDL statements to be issued within a user defined transaction, i.e. do not use statements such as CREATE, DROP and ALTER when the session’s transaction mode is kSessionTranManual. Do not use the $indexes() method using kSessionTranManual since this method creates a table.
12. Sybase automatically strips spaces from character data returned to Omnis.
13. “Data buffers could not be allocated” error following a logon attempt:  
    This error normally occurs if the Sybase environment variables; SYBASE, SYBASE\_OCS and/or LANG/LC\_ALL are not correct.  
    Check the sybinit.err file (in the Omnis folder) for more details about the error.

Further troubleshooting notes, “how-tos” and tips can be found on the Omnis website at: www.omnis.net/technotes

### Sybase Data Type Mapping

The following tables describe the data type mapping for Omnis and Sybase.

#### Omnis to Sybase

|  |  |
| --- | --- |
| **Omnis Data Type** | **Sybase Data Type** |
| **CHARACTER** |  |
| Character/National 0 | varchar(1) |
| Character/National 1 <= n <= 255 | varchar(n) |
| Character/National > 255 | text |
| **DATE/TIME** |  |
| Short date (all subtypes) | datetime |
| Short time | datetime |
| Date time (#FDT) | datetime |
| **NUMBER** |  |
| Short integer (0 to 255) | tinyint |
| Integer 32 bit | int |
| Integer 64 bit | bigint |
| Short number 0dp | numeric(9,0) |
| Short number 2dp | numeric(9,2) |
| Number floating dp | double precision |
| Number 0..14dp | numeric(15,0..14) |
| **OTHER** |  |
| Boolean | bit |
| Sequence | int |
| Binary/Picture/List/Row/Object/Item reference where $blobsize <= 255 | varbinary($blobsize) |
| Binary/Picture/List/Row/Object/Item reference where $blobsize > 255 | image |

#### Sybase to Omnis

|  |  |
| --- | --- |
| **Sybase Data Type** | **Omnis Data Type** |
| **CHARACTER** |  |
| char(n) | Character n |
| varchar(n) | Character n |
| nchar(n) | Character n |
| nvarchar(n) | Character n |
| text | Character 10,000,000 |
| **DATE/TIME** |  |
| datetime | Date time (#FDT) |
| smalldatetime | Date time (#FDT) |
| **NUMBER** |  |
| tinyint | Short integer (0 to 255) |
| smallint | Short number 0dp |
| int | Integer 32 bit |
| bigint | Integer 64 bit |
| numeric(p,n) | Number (n)dp |
| decimal(p,n) | Number (n)dp |
| real | Number floating dp |
| float | Number floating dp |
| double precision | Number floating dp |
| money | Number 4dp |
| smallmoney | Number 4dp |
| **OTHER** |  |
| bit | Boolean |
| binary(n) | Binary |
| varbinary(n) | Binary |
| image | Binary |
|  |  |

## DB2

This section contains the additional information you need to access a DB2 Universal Server database, including server-specific programming, data type mapping to and from DB2, as well as troubleshooting. For general information about logging on and managing your database using the Omnis SQL Browser, refer to the earlier parts of this manual.

### Properties and Methods

In addition to the “base” properties and methods documented in the *SQL Programming* chapter, the DB2 DAM provides the following additional features.

#### Session Properties

|  |  |
| --- | --- |
| Property | Description |
| $datetimeformat | This stores an Omnis date format string used to map a Date time (#FDT) bind variable to the correct server representation. This is necessary as DB2 supports different regional timestamp formats. The date is stored on the server in an internal binary representation.  The default format is 'y-M-D H:N:S'  This method is equivalent to the old-style <DATETIME\_FORMAT> keyword. |
| $drivername | The name of the session driver. |
| $driverodbcversion | The version number of the session driver. |

#### Session Methods

|  |  |
| --- | --- |
| Method | Description |
| $getdatasources() | SessionObj.$getdatasources(lListOrRow) populates the list with the name and description of the data sources defined on the client machine.  The list is redefined as having two columns- *DataSourceName* and *Description*. DataSourceName is defined as Character 32. Description is defined as Character 255.  This method is equivalent to the old-style **<**GET\_DATASOURCES**>** keyword. |

#### Statement Properties

|  |  |
| --- | --- |
| Property | Description |
| $erroronnodata | If set to kTrue (default), $execute() and $execdirect() will fail if execution returns SQL\_NO\_DATA, i.e. if a row addressed by the SQL statement could not be found. If set to kFalse, SQL\_NO\_DATA errors are ignored to be consistent with other databases. $erroronnodata does not affect SELECT statements. |

### Connecting to your Database

To connect to your database, you need to create a session object with a subtype of DB2SESS. In order to log on to the database using the SessObj.$logon() method, the hostname must contain the catalog database name entered using the DB2 Command Line Processor or using the Client Configuration Assistant if installed. The user name and password should contain the values required by the database. For example:

###### Do SessObj.$logon(‘MyDatabase’,‘UserID’,‘Password’,‘MySession’) Returns #F

In the event of connection failure, the DAM will timeout as dictated by any timeout policy in use by the server. Logon failures are usually reported immediately.

### Transactions

Generally, using manual transaction mode results in increased performance because the session object does not force a commit after each statement.

If you do not have a results set pending, the DB2 session object will commit each statement if the transaction mode is automatic. If the transaction mode is server, the server commits the statement automatically.

### Dates

The session property $defaultdate allows default values to be added to date values mapped to the server where the Omnis date value does not contain complete information, e.g. when a Short time is mapped to a server DATETIME. The date stored in this property is in a generic format, i.e. it is compatible with any regional date format that the server may be using.

### Boolean Type

DB2 does not include a specific type for storing single bit data. The Omnis Boolean type is therefore converted to a CHAR(3) value and stored as ‘YES’ or ’NO’ in the server table. The string representation can then be mapped back to an Omnis Boolean type when the data is retrieved.

### BLOB Type

The session property $blobsize can be used to specify the size argument for columns of type BLOB generated when the $createnames and $coltext methods are used.

Values range from 1 to 10000000. The default value for $blobsize is 10000000 which is also the maximum size of an Omnis binary variable.

This property is equivalent to the old-style <SETBLOBSIZE> keyword.

### Meta-Data Queries

The meta-data statement methods $columns(), $indexes() and $tables() allow you to receive information about the objects in your database. The $tables() method takes an optional owner name as a parameter. The $indexes() and $columns() methods optionally take the database and/or owner name with the table name parameter. See SQL Programming for more information on these methods.

When a database, owner or table name is specified, the result set is constrained by those schemas which meet the filter criterion, i.e. to return column information about the “addressbk” table owned by “robert” in database “acc\_db” the following can be issued.

Do tStatement.$columns('acc\_db.robert.addressbk') Returns #F

### DB2 Troubleshooting

**Reserved Words**

This section covers the DB2 specific reserved words.

The following schema names are reserved: SYSCAT, SYSFUN, SYSIBM & SYSSTAT.

In addition, it is strongly recommended that schema names never begin with the SYS prefix, as SYS is by convention used to indicate an area reserved by the system.

There are no words that are specifically reserved words in DB2. Keywords can be used as ordinary identifiers, except in a context where they could also be interpreted as SQL keywords. In such cases, the word must be specified as a delimited identifier. For example, COUNT cannot be used as a column name in a SELECT statement unless it is delimited.

IBM SQL and ISO/ANSI SQL92 include reserved words, these reserved words are not enforced by DB2 Universal Database, however it is recommended that they not be used as ordinary identifiers, since this reduces portability. Please see the final chapter in this manual which lists the SQL reserved words.

Further troubleshooting notes, “how-tos” and tips can be found on the Omnis website at: [www.omnis.net/technotes](http://www.omnis.net/technotes)

### DB2 Data Type Mapping

The following table describes the data type mapping for Omnis to DB2 connections. This mapping is predefined and is based on the best fit for each of the Omnis data types.

#### Omnis to DB2 UDB

|  |  |
| --- | --- |
| **Omnis data type** | **Server data type** |
| **CHARACTER** |  |
| Character/National(n) <= 4000 | VARCHAR(n) |
| 4000 < Character/National(n) <= 32,700 | LONG VARCHAR(n) |
| Character/National(n) > 32,700 | CLOB(n) |
| **DATE/TIME** |  |
| Short date (all subtypes) | DATE |
| Short time | TIME |
| Date time (#FDT) | DATETIME |
| **NUMBER** |  |
| Short integer | SMALLINT |
| Integer 32 bit | INTEGER |
| Integer 64 bit | BIGINT |
| Short number 0 dp | DOUBLE |
| Short number 2 dp | DOUBLE |
| Number 0..14 dp | DOUBLE |
| **OTHER** |  |
| Boolean | CHAR (3) |
| Sequence | INTEGER |
| Picture | BLOB($blobsize) |
| Binary | BLOB($blobsize) |
| List | BLOB($blobsize) |
| Row | BLOB($blobsize) |
| Object | BLOB($blobsize) |
| Item reference | N/A |

#### DB2 UDB to Omnis

|  |  |
| --- | --- |
| **Server Data Type** | **Omnis Data Type** |
| **NUMBER** |  |
| SMALLINT | Integer 32 bit |
| INTEGER | Integer 32 bit |
| BIGINT | Integer 64 bit |
| DECIMAL(p,s) | Number (s)dp |
| NUMERIC(p,s) | Number (s)dp |
| FLOAT | Number floating dp |
| REAL | Number floating dp |
| DOUBLE | Number floating dp |
| **CHARACTER** |  |
| CHAR(n) | Character (n) |
| VARCHAR(n) | Character (n) |
| LONG VARCHAR(n) | Character (n) |
| CLOB(n) | Character (n) |
| **DATE/TIME** |  |
| DATE | Short date |
| TIME | Short time |
| TIMESTAMP | Date time (#FDT) |
| **BINARY** |  |
| BINARY | Binary |
| VARBINARY | Binary |
| LONGVARBINARY | Binary |
| BLOB | Binary |
| **EXTENDERS** |  |
| IMAGE | Binary |
| AUDIO | Binary |
| VIDEO | Binary |
| TEXT | Binary |

## MySQL

This section contains the additional information you need to access a MySQL database, including server-specific programming, data type mapping to and from MySQL, as well as troubleshooting. For general information about logging on and managing your database using the Omnis SQL Browser, refer to the earlier parts of this manual.

### Properties and Methods

In addition to the “base” properties and methods documented in the *SQL Programming* chapter, the MySQL DAM provides the following additional features.

#### Session Properties

|  |  |
| --- | --- |
| Property | Description |
| $clientflags | SessObj.$clientflags sets the optional client flags logon parameter before executing $logon(), (The value can consist of several values added together if required). Their use is beyond the scope of this text and the default value of zero should be suitable for most purposes. Client flags are discussed further in the MySQL C API reference under mysql\_real\_connect(). |
| $database | SessObj.$database sets the additional *database* logon parameter before executing $logon(). Once logged on however, assigning a new value to this property causes the current database to change. When the session is created, a default value of “mysql” is assigned to this property. |
| $defaultdateisempty | If kTrue, fetched datetimes matching $defaultdate are treated as empty values. (Studio 8.0.2) |
| $hostinfo | SessObj.$hostinfo describes the type of connection in use, including the server host name. (Read-only) |
| $logontimeout | The number of seconds before a connection attempt times out. SessObj.$connectoption() can also be used to set a logon timeout if required. |
| $port | SessObj.$port sets the additional *port* logon parameter before executing $logon(). This will be the port number for a TCP/IP connection. The default port number is 3306. |
| $protoversion | SessObj.$protoversion is the version of the protocol in use by the current connection. (Read-only) |
| $socket | SessObj.$socket is the socket or named pipe that should be used for the connection, applicable to non-TCP/IP connections only. |
| $sslcipher | SessObj.$sslcipher returns the name of the SSL cipher being used for the current SSL connection or empty for a non-SSL connection. (Read-only) |
| $threadid | SessObj.$threadid is the thread ID of the current connection. (Read-only) |
| $threadsafe | SessObj.$threadsafe is kTrue if the client library was compiled as thread-safe. (Read-only) |

#### Session Methods

|  |  |
| --- | --- |
| Method | Description |
| $changeuser() | SessObj.$changeuser({cUsername,cPassword}) changes the current user. The new user will be connected to the database identified by the $database property. |
| $characterset() | SessObj.$characterset() returns the name of the default character set for the current connection. |
| $connectoption() | SessObj.$connectoption({iOption,vArgument}) specifies extra connect options and affects the behavior of a connection. This method may be called multiple times to set several options. Available option constants can be found in the Catalog (F9) under MYSQLDAM-ConnectOptions. For further details, refer to the MySQL C API documentation for mysql\_options(). As of Studio 8.0.2, $connectoption() can also be called with named connection attributes, e.g. Do sessObj.$connectoption(‘program\_name’,’My Program’) |
| $getdatatypemapping() | SessObj.$getdatatypemapping({lMappings}) retrieves a list of Omnis-to-MySQL data type mappings currently in use by the session. This list is formatted as described in the section below and is suitable for re-assignment to the session via the $setdatatypemapping() method if required. On successful retrieval of the list, $getdata typemapping() returns kTrue, otherwise kFalse is returned. $getdatatypemapping() can be called either before or after the session has logged on. |
| $insertid() | SessObj.$insertid() returns the ID of an AUTO\_INCREMENT column generated by the most recently executed query. Use this function after you have performed an INSERT query into a table that contains an AUTO\_INCREMENT field. $insertid() returns zero if the previous query did not generate an AUTO\_INCREMENT value, or was not an INSERT/UPDATE |
| $ping() | SessObj.$ping() checks whether the connection to the server is working. If it has gone down, an automatic reconnection is attempted. $ping() returns kTrue if the connection is alive. |
| $query() | SessObj.$query(cSqlText) allows SQL statements not supported by the MySQL prepared statement protocol to be executed directly on the connection. At the time of writing, such statements include the following administrative commands: GRANT, DROP DATABASE/USER, HANDLER, TRUNCATE, ALTER DATABASE/INDEX/USER, CREATE DATABASE/INDEX/USER, USE, LOCK TABLES, UNLOCK TABLES, SAVEPOINT, ROLLBACK TO SAVEPOINT, FLUSH, CACHE INDEX, LOAD INDEX INTO CACHE, KILL CONNECTION/QUERY, RESET. cSqlText can be either a single SQL statement or multiple statements and can contain square bracket notation if required. Bind variables are not supported; this functionality is provided by the statement object. $query() returns kTrue on success, otherwise kFalse. Error messages are returned via the session object. |
| $queryinfo() | SessObj.$queryinfo() retrieves a string providing information about the most recently executed query for any statement derived from the session, but only for certain INSERT, UPDATE and ALTER statements. For further information, refer to the MySQL C API documentation for mysql\_info(). |
| $queryresult() | SessObj.$queryresult(lResult) allows the result set generated by a previous call to $query() to be returned. lResult is a list variable which is cleared and redefined from the columns of the result set. The entire result set is placed into lResult and returned via a single call to $queryresult(). $queryresult() performs limited data type conversion on the various data types, recognising binary, integer and decimal numbers, defaulting to Character for all other types. This method has no effect if there is no result set pending on the session object.  As of Studio 10.1 $queryresult() can be called repeatedly to return multiple result sets. $queryresult() returns kFalse when there are no more result sets to return. |
| $serverdebuginfo() | SessObj.$serverdebuginfo() instructs the server to write some debug information to its error log. For this to work, the connected user must have the SUPER privilege. The log file name can be specified when the server is started by specifying --log-error[=filename] on the command line. |
| $servershutdown() | SessObj.$servershutdown() asks the database server to shut down. The connected user must have SHUTDOWN privileges. Note: no further confirmation is sought before severing the connection and shutting the server down. Returns kTrue if the server was successfully shutdown. |
| $serverstatus() | SessObj.$serverstatus({cInfo}) returns information about the server's current status, including the uptime in seconds and the number of running threads, questions, reloads, and open tables. |
| $setdatatypemapping() | SessObj.$setdatatypemapping({lMappings}) sets the Omnis-to-MySQL data type mappings for the session. The supplied list contains a prioritised list of mappings for Omnis data types and subtypes to their intended MySQL server data types. See the section below on the format of the mapping list. $setdata typemapping() can be used to map certain data type/subtypes to custom MySQL data types, e.g. SET and ENUM types. This is also explained in the section below. On successful execution of $setdatatypemapping() kTrue is returned, otherwise kFalse is returned. SessObj.$setdata typemapping() can be called either before or after the session has logged on. |
| $sslset | SessObj.$sslset([cKey, cCert, cCA, cCAPath, cCipher]) is used for establishing a secure connection using SSL. It must be called before $logon(). cKey is the path name to the key file. cCert is the path name to the certificate file. cCa is the path name to the certificate authority file. cCAPath is the path name to a directory that contains trusted SSL CA certificates in pem format. cCipher is a list of permissible ciphers to use for SSL encryption. Any unused SSL parameters will be treated as NULL. |

#### Statement Methods

|  |  |
| --- | --- |
| Method | Description |
| $columns() | StatObj.$columns({cTableName}). Returns information describing the columns of the supplied table name. cTableName can be qualified with an optional database name; [database.]tablename if required.  The DamInfoRow column returned by $columns() contains additional information for each column described. The row is defined with the following columns:  UniqueKey kTrue if col is a unique index MultipleKey kTrue if col is part of a compound index Unsigned   kTrue if col has the UNSIGNED attribute ZeroFill       kTrue if col has the ZEROFILL attribute Binary       kTrue if col contains binary (BLOB/TEXT) data AutoIncrement  kTrue if col has the AUTO\_INCREMENT attribute Number    kTrue if col contains numeric data DefaultValue  Returns the default value for col as char data |
| $rpcprocedures() | StatObj.$rpcprocedures([cOwner]) generates a result set containing the names of stored procedures and functions which (optionally) were created by the named user. The DamInfoRow column returned by $rpcprocedures() contains additional information for each procedure described. The row is defined with the following columns:  Type   Specifies whether the row describes a procedure or   function Specific Name The specific name of the procedure. Language  The programming language contained within the procedure. SQL Data Access  Describes data usage characteristics of the procedure. Deterministic  kTrue if the procedure is ‘deterministic’, i.e. always produces the same result for the same input parameters. Security Type  Describes the permissions used when executing the procedure. Param List  Contains a comma separated list of input/output parameters. Returns  Describes the data type returned by a function. Body  Returns the text content of the procedure. Created  The date and time when the procedure was created. Modified  The date and time when the procedure was last modified. SQL Mode  Describes the SQL syntax supported by the procedure. Comment  User comment added when the procedure was created.  Stored procedures and functions are not supported in versions of MySQL prior to 5.0. |

### Logging on to MySQL

The MySQL DAM interfaces directly with the MySQL client library, therefore the way the DAM logs on to the server is slightly different to the other Object DAMs.

Specifically, the $logon() *hostname* parameter is taken as the server hostname or IP address. The username and password parameters are supplied as normal.

As well as the parameters supplied to $logon(), there are some additional parameters which you can set using the following session properties:

1. $port - The port number of the MySQL server
2. $database - The database name
3. $clientflags - Sets additional behavior for the logon
4. $socket - Specified if you do not want to use a TCP/IP connection

When logging on using the SQL Browser, default values are used when the Port and Database fields are left blank.

### Transactions

If you require transaction support with MySQL, your server needs to support BDB or InnoDB table types. Manual transactions may only be made on tables of these types.

When creating tables, you need to specify the table type required if you do not want the default type (MyISAM).

The following properties and methods apply to transactions. Use of these properties or methods is equivalent to executing the SQL statements shown:

|  |  |
| --- | --- |
|  |  |
| $begin() | SessObj.$begin() = Begin |
| $commit() | SessObj.$commit() = Commit |
| $rollback() | SessObj.$rollback() = Rollback |
| $transactionmode | SessObj.$transactionmode.$assign(SessionMode)  kSessionTranAutomatic: Autocommit = 1 (the default) kSessionTranServer: Autocommit = 1  kSessionTranManual: Autocommit = 0 |

If you are not using InnoDB or BDB table types, you can achieve table locking using the MySQL lock tables or unlock tables SQL commands. Refer to the MySQL language reference for further details.

### MySQL Data Type Mapping

#### Omnis to MySQL

The default data type mappings from Omnis to MySQL are shown below.

|  |  |
| --- | --- |
| Omnis Date Type | MySQL Data Type |
| **CHARACTER** |  |
| Character n (n<=255) | VARCHAR(n) |
| National n (n<=255) | NATIONAL VARCHAR(n) |
| Character/National n (n<=65534) | TEXT |
| Character/National n (65534<n<=10000000) | MEDIUMTEXT |
| **NUMBER** |  |
| Integer 64 bit | BIGINT |
| Integer 32 bit | INT |
| Short integer | TINYINT UNSIGNED |
| Number 0..14dp | DECIMAL(15,0..14) |
| Number floating dp | DOUBLE |
| Short number 0/2dp | DECIMAL(9,0/2) |
| **DATE/TIME** |  |
| Short date (all subtypes) | DATE |
| Short time | TIME |
| Datetime (#FDT) | DATETIME |
| **OTHER** |  |
| Boolean | BOOL |
| Picture | MEDIUMBLOB |
| List | MEDIUMBLOB |
| Row | MEDIUMBLOB |
| Object | MEDIUMBLOB |
| Binary | MEDIUMBLOB |
| Item reference | TINYBLOB |
| Sequence | INT UNSIGNED AUTO\_INCREMENT PRIMARY KEY |

Note that this is equivalent to the list returned by a call to $getdatatypemapping() on a newly created session object:

|  |  |  |  |
| --- | --- | --- | --- |
| OmnisType | OmnisSubtype | Parameter | MySqlType |
| char | simple | 255 | VARCHAR($) |
| char | national | 255 | NATIONAL VARCHAR($) |
| char | national | 65534 | TEXT |
| char | national | 10000000 | MEDIUMTEXT |
| integer | 64 bit | 0 | BIGINT |
| integer | 32 bit | 0 | INT |
| integer | shortint | 0 | TINYINT UNSIGNED |
| number | 14dp | 0 | DECIMAL(15,$) |
| number | float | 0 | DOUBLE |
| number | 2dpShortnum | 0 | DECIMAL(9,$) |
| boolean |  | 0 | BOOL |
| date | date2000 | 0 | DATE |
| date | time | 0 | TIME |
| date | datetime | 0 | DATETIME |
| picture |  | 0 | MEDIUMBLOB |
| list |  | 0 | MEDIUMBLOB |
| row |  | 0 | MEDIUMBLOB |
| object |  | 0 | MEDIUMBLOB |
| binary |  | 0 | MEDIUMBLOB |
| itemref |  | 0 | TINYBLOB |

#### Assigning a new mapping table using $setdatatypemapping()

If you need to make a change to the default Omnis to MySQL data type mappings, you should probably base your new mappings on the default mappings obtainable by calling $getdatatypemapping() and add/remove lines to the returned list as required before calling $setdatatypemapping() to install the new mapping table.

There are a number of points to note regarding the format and processing of the list used by these methods and these are discussed below.

**Omnis subtype precedence**

Note that where multiple occurrences of Omnis types appear in the list, the Omnis subtype(s) should be specified in ascending *numerical* order, especially if you intend a mapping to apply to all Omnis subtypes <= the supplied value. This is because when searching for a match, the list is processed in order from the first entry to last- the search ends at the first matching entry. The full list of acceptable data subtypes can be found in the Omnis catalog (F9) and their text equivalents are shown in the OmnisSubtype column, as summarised below:

|  |  |  |
| --- | --- | --- |
| Omnis constant | Numeric value (precedence) | Character equivalent (OmnisSubtype) |
| **Character subtypes** |  |  |
| kSimplechar | 0 | simple |
| kNatchar | 1 | national |
| **Integer subtypes** |  |  |
| k32bitint | 0 | 32 bit integer |
| kShortint | 32 | shortint |
| k64bitint | 64 | 64 bit integer |
| **Number subtypes** |  |  |
| k0dp | 0 | 0dp |
| k1dp | 1 | 1dp |
| k2dp | 2 | 2dp |
| k3dp | 3 | 3dp |
| k4dp | 4 | 4dp |
| k5dp | 5 | 5dp |
| k6dp | 6 | 6dp |
| k8dp | 8 | 8dp |
| k10dp | 10 | 10dp |
| k12dp | 12 | 12dp |
| k14dp | 14 | 14dp |
| kFloatdp | 24 | float |
| k0dpShortnum | 32 | 0dpShortnum |
| k2dpShortnum | 34 | 2dpShortnum |
| **Datetime subtypes** |  |  |
| kDate1900 | 0 | date1900 |
| kDate1980 | 1 | date1980 |
| kDate2000 | 2 | date2000 |
| kTime | 6 | time |
| kDatetime | 1000 | Datetime |

Note that where an Omnis type does not have a subtype (e.g. Binary), it is acceptable to leave the subtype column blank.

**Parameter column**

The ‘Parameter’ value in the data type mapping list specifies the maximum length or size of data to which the mapping will apply, e.g. a value of 255 specified for a character data type signifies that that mapping will apply to Omnis character data with a length of <= 255 characters. You should therefore ensure that where there are multiple occurrences of the same Omnis data type and subtype, these are entered in ascending order of parameter value. The maximum size of an Omnis character/binary field is 10,000,000 bytes.

**MySqlType column**

The MySqlType value specifies the string which will be returned by session.$createnames() when that row matches the supplied Omnis data type & subtype. This can be (but is not restricted to) any string which constitutes a valid MySQL column type, e.g.

###### SET(‘One’,‘Two’,‘Three’,‘Four’)

Where a ‘$’ character is used as part of the MySQL type, the appropriate length or scale attribute will be substituted when $createnames() is called.

For valid MySQL data type assignments, the DAM also uses the data type mapping table to map outgoing bind variables to their corresponding MySQL column types.

**Example applications of $setdatatypemapping()**

The intended use of $setdatatypemapping() is to allow schema columns to conditionally map to custom MySQL data types, not implemented by default, e.g. the SET, ENUM, GEOMETRY and YEAR types and also to allow extra type qualifiers to be added, such as UNSIGNED, PRIMARY KEY, AUTO\_INCREMENT, etc.

To get $createnames() to return one of these types, you might for example isolate a specific character string length and add a data type mapping for your new type.

Then whenever $createnames() encounters a string of that specific length, the mapping to your new type will occur. If you want to implement the SET data type, you could insert a new mapping entry between the first and second default entries, adjusting the length parameters as shown below:

|  |  |  |  |
| --- | --- | --- | --- |
| char | simple | 254 | VARCHAR($) |
| char | simple | 255 | SET(‘One’,’Two’,’Three’,’Four’) |
| char | national | 255 | NATIONAL VARCHAR($) |

Alternatively, you could dedicate the ‘national’ character subtype for your custom data types, leaving the ‘simple’ character subtype for standard character mappings.

#### MySQL to Omnis

The following mappings are hard-coded and cannot be altered.

|  |  |  |
| --- | --- | --- |
| MySQL Column Type | Range | OmnisType/ Subtype |
| **NUMBER** |  |  |
| BIT/BOOL/TINYINT(1) |  | Boolean |
| TINYINT | (-128..+127) | Integer 32 bit |
| SMALLINT | 2^16 (-32768..+32767) | Integer 32 bit |
| MEDIUMINT | 2^24 (-8388608..+8388607) | Integer 32 bit |
| INT/INTEGER | 2^32 (-2147483648..+2147483647) | Integer 32 bit |
| BIGINT | 2^64 (-2^63..+2^63-1) | Integer 64 bit |
| FLOAT |  | Num floating dp |
| DOUBLE/REAL |  | Num floating dp |
| DEC/DECIMAL/NUMERIC | (precision<=9 & scale<=2) | ShortNum 0..2dp |
| DEC/DECIMAL/NUMERIC | (precision>9 or scale>2) | Num 0..14dp |
| **DATE/TIME** |  |  |
| DATE |  | Datetime (#FDT) |
| DATETIME |  | Datetime (#FDT) |
| TIMESTAMP |  | Datetime (#FDT) |
| TIME |  | Datetime (#FDT) |
| YEAR |  | Integer 32 bit |
| **CHARACTER** |  |  |
| CHAR | 1 to 255 bytes fixed | Character |
| VARCHAR | 0 to 255 bytes varying | Character |
| TINYTEXT | 255 (2^8 – 1) bytes | Character |
| TEXT | 65535 (2^16 – 1) bytes | Character |
| MEDIUMTEXT | 16777215 (2^24 – 1) bytes | Character |
| LONGTEXT | 4294967295 (2^32 – 1) bytes | Character |
| ENUM |  | Character |
| SET |  | Character |
| **BINARY** |  |  |
| TINYBLOB | 255 (2^8 – 1) bytes | Binary |
| BLOB | 65535 (2^16 – 1) bytes | Binary |
| MEDIUMBLOB | 16777215 (2^24 – 1) bytes | Binary |
| LONGBLOB | 4294967295 (2^32 – 1) bytes | Binary |

### MySQL Troubleshooting

The following points may help in resolving programming issues encountered using MySQL session and statement objects. For additional updated trouble shooting issues, refer to the readme file which accompanies your installation media.

1. $sqlstripspaces has no effect for MySQL sessions. MySQL automatically strips trailing spaces from data inserted into CHAR and VARCHAR columns. Character data returned from the server will already be stripped of trailing spaces.
2. MySQL 4.1 does not support chunking of fetched (output) LOB data (TEXT and BLOB types). Chunking of input LOB data is supported.
3. $rowcount will be –1 following execution of a SELECT, SHOW or EXPLAIN statement. This is because MySQL cannot determine this value until the final row has been fetched.
4. The MySQL DAM is compatible with MySQL Server version 4.1 and later. Prior to Omnis Studio version 4.1, connection is only possible to a commercial version of MySQL Server (i.e. MySQL “pro” or “classic”). Later versions of Studio do not have this restriction.

Further troubleshooting notes, “how-tos” and tips can be found on the Omnis website at: www.omnis.net/technotes

## PostgreSQL

This section contains the additional information you need to access a PostgreSQL database, including server-specific programming, data type mapping to and from PostgreSQL, as well as troubleshooting. For general information about logging on and managing your database using the Omnis SQL Browser, refer to the earlier parts of this manual.

For additional information on changes to the PostgreSQL DAM, refer to the readme file which accompanies your installation media.

### PostgreSQL Client Libraries

This section discusses the PostgreSQL client library, which must be present on the library search path before the PostgreSQL DAM can be used.

#### Win32 platforms

For Win32 platforms, the library search path includes the Windows\System32 folder or any location in the PATH environment variable, including the folder containing omnis.exe. The Win32 client library is named **libpq.dll**.

#### Linux and macOS platforms

The Unix ports of the PostgreSQL DAM look for **libpq.so** or **liqpq.dylib** under Linux and macOS respectively.

In most cases the library present on your system will be labelled according to the version you have installed. For example on Linux, libpq.so might be a symbolic link to the target library libpq.so.5.0. A detailed directory listing shows this relationship, e.g.

###### 12 2007-01-15 10:20 libpq.so -> libpq.so.5.0 117338 2007-01-15 10:20 libpq.so.5.0

Under Linux and macOS therefore, it is essential that the target library and symbolic link to it both exist either in the library search path or in the same folder as the Omnis executable.

### Properties and Methods

In addition to the “base” properties and methods documented in the *SQL Programming* chapter, the PostgreSQL DAM provides the following additional features.

#### Session Properties

|  |  |
| --- | --- |
| Property | Description |
| $maxvarchar | Defines the maximum size above which- Omnis Character fields will be mapped to TEXT type instead of VARCHAR. The default value for this property is 2000. |
| $database | Used to set the additional dbname logon parameter. If not specified, defaults to be the same as the user name. |
| $service | Service name to use for additional parameters. It specifies a service name in pg\_service.conf that holds additional connection parameters. This allows applications to specify only a service name so connection parameters can be centrally maintained. |
| $protocolversion | (Read-only)This property reports the communication protocol version supported by the client library. DAMPGSQL requires version 3.0 or higher in order to work correctly. |
| $backendpid | (Read-only) Following logon, this property holds the process ID of the backend server process handling the connection. This may be useful for debugging purposes since the PID is reported in NOTIFY messages. |
| $port | Used to set the additional port logon parameter. This property has a default value of 5432. |
| $socket | (Read-only) Following logon, this property holds the file descriptor number of the connection socket to the server. A valid descriptor will be greater than or equal to 0; a result of -1 indicates that no server connection is currently open. |
| $options | Used to specify additional text to be appended to the logon connection string. One or more parameter settings can be added, separated by spaces. The options string is limited to 255 characters. Discussion of advanced connection options is beyond the scope of this text but an example string might be:  Do sess.$options.$assign(“options='-c geqo=off' sslmode=require”) |
| $logontimeout | Maximum wait for a connection, in seconds. Zero implies wait indefinitely. The default timeout is set to 15 seconds. A timeout of less than 2 seconds is not recommended. |
| $timezone | Character string representing the time zone to be appended on to bind variables being inserted into TIMETZ and TIMESTAMPTZ columns. The default time zone is “+00” but $timezone will accept any character string (80 characters max). |
| $usetimezone | If set to kTrue, the value contained in $timezone is appended to outgoing Time and Datetime bind variables. This property also affects the text returned by $createnames() for Time and DateTime columns.  $timezone will be ignored during insert/update of TIMESTAMP & TIME columns |
| $serializable | If set to kTrue, manual transactions will be created using the Serializable isolation level. When set to kFalse (the default), manual transactions will be created using the Read Committed isolation level. |
| $readonly | If set to kTrue, manual transactions will be created using read-only access mode. When set to kFalse (the default), transactions will have read/write access. |
| $schema | The optional schema name to be prepended to table names. Used by the SQL Browser when performing SELECTs. The default schema name is an empty string. |
| $numericprecision | Defines the precision used by $createnames() when mapping Omnis number (dp) columns to the NUMERIC type. Cannot be set lower than the default value: 15. |
| $sequencetoint | If set to kTrue, the Omnis Sequence type is mapped to INTEGER. If set to kFalse (the default), the Sequence type is mapped to SERIAL. Affects $createnames() and outward bind variables. |
| $char38touuid | If set to kTrue, Omnis character types of field length 38 are mapped to the PostgreSQL 8.3 Universally Unique Identifier type (UUID). |
| $char39tooid | If set to kTrue, Omnis character types of field length 39 are mapped to the PostgreSQL Object Identifier type (OID). |
| $defaultdateisempty | If kTrue, fetched datetimes matching $defaultdate are treated as empty values. |
| $programname | If specified, registers an application name during $logon() which will be stored in the pg\_stat\_activity table. |
| $infinitydates | If kTrue, date value 31 Dec 9999 and datetime value 31 Dec 9999 23:59:59 maps to the special value; ‘infinity’. |

#### Session Methods

|  |  |
| --- | --- |
| Method | Description |
| $connectstatus() | Returns a PGSQLDAM Connection Status constant representing the current state of the connection to the database server, or empty if not connected. |
| $escapebinary() | Returns a text-escaped representation of the supplied binary variable, suitable for use in an SQL statement as a quoted string literal. The returned string does not include the quotes. |
| $getssl() | Returns qtrue if the connection is using SSL, qfalse otherwise. An optional list parameter can also be passed to return additional information. Currently, the SSL type and version are returned. |
| $listen() | Listens for the specified notification channel name and calls obj.$notify() when triggered. Call $unlisten() to remove the listener. |
| $notify() | Create this method inside an object class of subtype PGSQLDAM.PGSQLSESS. obj.$notify() will be called with a single parameter of type *row* when a client issues a NOTIFY command with a channel name previously registered using the $listen() method. The row parameter will be defined as: Channel: the notification channel name PID: the ID of the calling client process Message: character variable containing the message ‘payload’ |
| $transactionstatus() | Returns the current in-transaction status of the server. The status can be kPgSqlTranIdle (currently idle), kPgSqlTranActive (a command is in progress), kPgSqlTranInTrans (idle, in a valid transaction block), or kPgSqlTranINError (idle, in a failed transaction block). kPgSqlTranUnknown is reported if the connection is bad. kPgSqlTranActive is reported only when a query has been sent to the server and not yet completed. |
| $parameterstatus() | Looks up a current parameter setting of the server. Supported (string) parameters include server\_version, server\_encoding, client\_encoding, is\_superuser, session\_authorization, DateStyle, TimeZone, integer\_datetimes, and standard\_conforming\_strings. For a full list, refer to the API documentation for the PQparameterStatus function. |
| $reset() | Resets the communication channel to the server.  This function will close the connection to the server and attempt to re-establish a new connection to the same server, using all the same parameters previously used. This may be useful for error recovery if a working connection is lost. |
| $cancel() | Requests that the server abandon processing of any transactions pending on the session. Successful execution is no guarantee that the request will have any effect, however. If the cancellation is effective, the current command(s) will terminate early and return an error result. |
| $addcustomtype() | $addcustomtype(iFieldlength,cDatatype) Creates a custom data type mapping for specified Omnis character subtypes. Intended to allow creation and insertion into PostgreSQL 8.3 enum, xml and json columns. |
| $clearcustomtypes() | $clearcustomtypes() Removes all previously created custom data type mappings. |
| $lobimport() | $lobimport(cFilename[, iOid]) Imports the contents of the specified operating system file into the database and returns the new OID on success, zero otherwise. If a specific OID value is desired, it may be passed in via parameter 2. Must be called within a manual transaction block. |
| $lobexport() | $lobexport(cFilename, iOid) Exports the object specified by iOid into the specified operating system file. Must be called within a manual transaction block. Returns kTrue on success. |
| $lobcreate() | $lobcreate([iOid]) Creates a new large object and returns the new OID value on success, zero otherwise. If a specific OID value is desired, it may be passed in via parameter 1. Must be called within a manual transaction block. |
| $lobunlink() | $lobunlink(iOid) Removes the specified object from the database and unlinks the Object Identifier, effectively deleting the object. Must be called within a manual transaction block. Returns kTrue if the object was successfully unlinked. |
| $lobopen() | $lobopen(iOid[, bReadOnly]) Opens the specified large object for reading/writing and returns the large-object descriptor which is only valid for the duration of the current transaction. If bReadOnly is specified (kTrue), a read-only snap shot of the object is taken as it was at the start of the transaction. |
| $lobwrite() | $lobwrite(iDesc, xBinary[, iSize]) Writes the supplied binary data to the specified large-object descriptor, returning the number of bytes that were written on success, or -1 on failure. By default, the entire binary field is written unless iSize is specified. |
| $lobread() | $lobread(iDesc, xBinary[, iSize]) Reads the large object specified by iDesc into xBinary and returns the number of bytes read on success, or -1 on failure. If specified, iSize bytes are allocated and read from the large object. If omitted, *$blobsize* bytes are allocated/requested. |
| $lobseek() | $lobseek(iDesc, iOffset, iWhence) Moves the read/write pointer within an open large object by *iOffset* bytes. iWhence governs how the offset is interpreted; kPgSqlSeekSet specifies an absolute offset from the start of the object, kPgSqlSeekCur specifies an offset from the current position, kPgSqlSeekEnd specifies an offset from the end of the object. Returns the new location on success, or -1 on failure. |
| $lobtell() | $lobtell(iDesc) Returns the current position of the read/write pointer within the large object specified by iDesc, or -1 on failure. |
| $lobtruncate() | $lobtruncate(iDesc, iSize) Resizes the specified large object to *iSize* bytes. If iSize is larger than the current size, the large object is padded with null bytes. Returns kTrue on success, kFalse otherwise. |
| $lobclose() | $lobclose(iDesc) Explicitly closes the specified large-object descriptor. Any large-object descriptors that remain open at the end of a transaction will be closed automatically. Returns kTrue on success, kFalse otherwise. |
| $unlisten() | Removes the specified notification channel from the listener queue. |

#### Statement Properties

|  |  |
| --- | --- |
| Property | Description |
| $sqlstate | (Read only) On error, this property contains the five-character SQLSTATE associated with the $nativeerrortext. Refer to the PostgreSQL reference manual for a full list of SQLSTATEs. |

### Logging on to PostgreSQL

In addition to the hostname, username and password parameters provided by the $logon() method, the PostgreSQL DAM provides several session properties which enable additional logon parameters to be set. These should be set before calling $logon().

1. $database is used to specify the dbname connection parameter.
2. $port is used to specify the port connection parameter.
3. $logontimeout is used to specify the connect\_timeout parameter.
4. $options is used to specify further optional connection parameters.
5. $service is used to specify a service (filename) to use for additional parameters.

### Metadata Functions

1. $indexes()

The DamInfoRow for $indexes() is defined with a single column containing the SQL text used to define the index.

1. $tables()

The PostgreSQL DAM implements $tables() slightly differently. In particular- only the kStatementServerTable and kStatementServerView parameters are supported. This is because the processes for querying tables are incompatible with those for querying views. (kStatementServerAll defaults to kStatementServerTable).

The DamInfoRow for $tables() is defined with three Boolean columns with additional information on the table or view: HasIndexes, HasRules & HasTriggers.

### Transactions

PostgreSQL supports two transaction isolation levels: Read Committed (the default) and Serializable. Using Read Committed mode, a statement can only see rows that were committed before the current transaction began. Using Serializable mode, all statements in the current transaction can only see rows that were committed before the first query or data-modification statement was executed in this transaction.

Transactions can also be instantiated as read-only if required. This enables significant performance improvements for read operations. When a transaction is read-only, the following SQL commands are disallowed: INSERT,UPDATE,DELETE and COPY FROM if the table they would write to is not a temporary table; all CREATE,ALTER and DROP commands; COMMENT,GRANT,REVOKE,TRUNCATE,EXPLAIN ANALYZE and EXECUTE if the command they would execute is among those listed. Please refer to the PostgreSQL documentation on transactions for further details.

When using manual transaction mode (kSessionTranManual), the transaction isolation level can be switched between Read Committed and Serializable using the $serializable session property.

The access mode can be changed using the $readonly session property.

The PostgreSQL DAM treats the kSessionTranAutomatic and kSessionTranServer transaction modes identically. In either of these modes the server automatically begins and commits read/write transactions.

### Remote Procedure Calls

Note that PostgreSQL does not support the concept of stored procedures but supports functions instead. This has a few implications as described below.

1. $rpcprocedures(). The DamInfoRow returned by $rpcprocedures is defined with the following columns:

|  |  |
| --- | --- |
|  |  |
| Language | The implementation language or call interface for this function. |
| IsAgg | kTrue if this is an aggregate function. |
| SecDef | kTrue if this function is a security definer (i.e. a "setuid" function). |
| IsStrict | kTrue if this is a "strict" function. Strict functions must be prepared to handle null inputs. |
| RetSet | kTrue if the function returns a result set (i.e. multiple values of the specified data type). |
| Volatile | Indicates whether the function result depends only on its input arguments, or is affected by outside factors. It is i for "immutable" functions, which always deliver the same result for the same inputs. It is s for "stable" functions, whose results (for fixed inputs) do not change within a scan. It is v for "volatile" functions, whose results may change at any time, that have side-effects for other functions or tables or functions which cannot otherwise be optimised. |
| Source | Indicates how the function should be invoked. It might be the actual source code of the function for interpreted languages, a link symbol, a file name, or just about anything else, depending on the implementation language/call convention. |

1. $rpc(). Calling $rpc() is similar to executing a SQL SELECT statement of the form:

###### SELECT \* from proc\_name (param1, param2, … )

with the exception that $rpc() will also set any InputOutput or Output parameters.

Any return value generated by the function will be available via $rpcreturnvalue although in the case where the function generates a result set, it may be preferable to retrieve the entire set by calling $fetch(). The value returned by $rpcreturnvalue is also returned as the first row of this result set.

### Notification Channels

PostgreSQL supports asynchronous notification channels via its LISTEN, UNLISTEN and NOTIFY SQL commands. You can register the session object as a listener for a notification channel using the $listen() method, specifying the channel name to listen for. $unlisten() removes the listener.

Once registered, if any client executes a NOTIFY for that channel name, the listener calls the session object’s $notify() method with parameters that indicate the channel name, the notifier’s process ID and an optional ‘payload’/ text string.

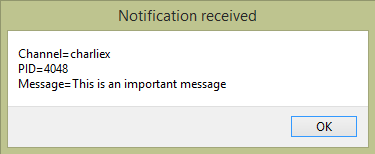
In the following example, oPgSQL is an object class with $superclass .PGSQLDAM.PGSQLSESS:

###### Do oPgSQL.$logon('192.168.0.96','postgres','postgres','session1') Returns #F Do oPgSQL.$listen('charliex') Returns #F

###### Do oPgSQL.$newstatement() Returns statObj Do statObj.$execdirect("notify charliex,'This is an important message'") Returns #F

Create method oPgSQL.$notify() with a single parameter of type Row to be called whenever a notification is received. For example:

###### OK message Notification received {Channel=[pRow.Channel]//PID=[pRow.PID]//Message=[pRow.Message]}



As of Studio 10.2, $listen() automatically encloses the channel name in double quotes when $quotedidentifier is kTrue.  
Unquoted channel names containing illegal characters now cause $listen() to return kFalse with $nativeerrortext; “Malformed unquoted identifier”.

Valid unquoted identifiers are case-insensitive, commence with a-z or \_ and can contain a-z, 0-9 as well as \_ and $. Quoted identifiers are case-sensitive and can contain any characters. PostgreSQL identifiers have a maximum length of 63 characters.

### Handling Dates

When kTrue, the $defaultdateisempty tells the DAM to convert retrieved datetimes to empty when they match $defaultdate.

### UUID, ENUM and XML Column Types

Support for the following data types is available in Omnis Studio 4.3.1 and above.

**UUID**

The PostgreSQL DAM is able to read and write Universally Unique Identifiers. An example of a UUID in standard form might be:

###### a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11 (36 characters)

but the DAM also accepts UUIDs formatted without hyphens and/or encapsulated using curly braces.

Output from UUID columns is always in the standard form.

To allow input binding of UUIDs and to make $createnames() return UUID types, it is necessary to set $char38touuid to kTrue. Once set, the Omnis Character 38 data subtype maps to UUID.

Note: there is no facility either in the PostgreSQL client library or in the DAM to create UUID values. This must be implemented by the Omnis application.

**ENUM**

Enumerated types are created by executing CREATE TYPE statements, for example:

###### Do statObj.$execdirect("CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy')") Returns #F ##creates the enumerated type

To make Omnis map certain character sub types to ENUMs, the $addcustomtype() method is provided.

The following example maps the Omnis Character 2001 data subtype to the “mood” enumerated type:

###### Do sessObj.$addcustomtype(2001,'mood') Returns #F

Once set, this mapping affects the text generated by $createnames() as well as input binding.

To clear previously defined enumerated type mappings, the $clearcustomtypes() method is provided.

**XML**

The $addcustomtype() method can also be used to force an Omnis Character subtype to map to the XML data type, for example:

###### Do sessObj.$addcustomtype(10001,'xml') Returns #F

As above, this mapping affects the text generated by $createnames() as well as input binding and remains in effect until $clearcustomtypes() is called.

### Large Object Support

As of Omnis Studio 5.1.1, the PostgreSQL DAM supports additional session object methods for manipulating large objects stored in the database.

The methods; $lobimport(), $lobexport() and $lobunlink() complement their SQL equivalents (lo\_import(), lo\_export() & lo\_unlink()) with the exception that the client-side methods operate on files in the client machine’s file system. The SQL functions operate on files in the database server’s file system. In other respects, the operation of these methods is comparable:

* $lobimport() – creates a large object and imports data into it from a local file.
* $lobexport() – retrieves data from a large object and writes it to a local file.
* $lobunlink() – removes a large object from the database.

There is an additional client-side method for creating a large object:

* $lobcreate() – creates a new (empty) large object and returns the OID value.

The large objects are identified by their OID values, which can subsequently be stored and retrieved in database *Oid* columns in a similar fashion to standard integers.  
Once created, the following methods can be used to manipulate data inside large objects:

* $lobopen() – opens a large object (OID) and returns a large object descriptor.
* $lobclose() – closes a large object descriptor.
* $lobread() – reads zero or more bytes from a descriptor into a Binary variable.
* $lobwrite() – writes zero or more bytes to a descriptor from a Binary variable.
* $lobseek() – repositions the read/write pointer within a large object.
* $lobtell() – reports the current pointer position within a large object.
* $lobtruncate() – resizes a large object to the desired size (in bytes).

To use these methods, it is important to note that large object operations must be performed within a **single transaction**, i.e. in manual transaction mode. Any open large object descriptors are automatically closed upon $commit(). For example:

###### Do cSess.$transactionmode.$assign(kSessionTranManual) Do cSess.$begin() Do cSess.$lobcreate() Returns lOid ## create new oid Do cSess.$commit()

###### Do cSess.$begin() Do cSess.$lobopen(lOid) Returns fileDesc Calculate lBinary as 'Some Unicode character data' Do cSess.$lobwrite(fileDesc,lBinary) Returns lNumBytes ## write data into the large object Do cSess.$lobseek(fileDesc,8,kPgSqlSeekSet) Returns lFilePos ## move the read/write pointer to byte 8/character position 3 Do cSess.$lobread(fileDesc,lCharValue) Returns lNumBytes ;;=> 'me Unicode character data' Do cSess.$commit() ## commit closes the descriptor

For further information on the behavior of these methods and the parameter values that may be applied, please refer to the Session Methods section above.

### JSON Column Types

As of Studio 8.0.3 you can select and insert JSON strings into PostgreSQL JSON and JSONB columns. In earlier versions of Studio, there was partial support for insertion of JSON strings and it was possible to select from JSON columns using the CAST(… as VARCHAR(n)) operator.

The client library will parse and validate text before insertion into JSON/JSONB columns. You can optionally use this feature to validate JSON strings, for example:

###### Do cStat.$execdirect("select '5'::json as myCol") Returns #F Do cStat.$fetchinto(lResult) ## Returns 5

###### Do cStat.$execdirect("select '{""col1"":1,""col2"":""mostly cloudy"",""col3"":true}'::json as myJSON") Returns #F Do cStat.$fetchinto(lResult) ## Returns {"col1":1,"col2":"mostly cloudy","col3":true}

###### Do cStat.$execdirect("select '{""col1"":1,""col2"":""600 meters"",""col3"":[[""one"",""two"",""three""]}'::json as myValue") Returns #F Do cStat.$fetchinto(lResult) ## Returns {"col1":1,"col2":"600 meters","col3":["one","two","three"]}

Any of the resulting strings can then be inserted into the database:

###### Do cStat.$execdirect('create table jsontest(col1 int, col2 jsonb)') Returns #F Do cStat.$execdirect('insert into jsontest values(1,@[lResult])') Returns #F

Note that JSON string literals must be suitably *escaped* in respect of quotes and square brackets, as shown above.

To insert JSON strings using bind variables, the $addcustomdatatype() method should be used. This tells the DAM to map Omnis character fields of a specific fieldlength to JSON and/or JSONB columns, and also allows $createnames() to generate JSON and/or JSONB columns. For example:

###### Do cSess.$addcustomtype(1000,'JSON') Returns #F Do cSess.$addcustomtype(1001,'JSONB') Returns #F Do lList.$definefromsqlclass('scTest') Calculate lSql as cSess.$createnames(lList)

Where scTest defines Character columns of length 1000 or 1001, the ‘JSON’ and ‘JSONB’ column type will be returned accordingly.

### PostgreSQL Data Type Mapping

#### Omnis to PostgreSQL

|  |  |
| --- | --- |
| Omnis Data Type | PostgreSQL Data Type |
| **CHARACTER** |  |
| Character/National n (n<=$maxvarchar) | VARCHAR(n) [4] |
| Character/National n (n>$maxvarchar) | TEXT |
| Character(38) | UUID [3] |
| **NUMBER** |  |
| Integer 64 bit | BIGINT |
| Integer 32 bit | INTEGER |
| Short integer | SMALLINT |
| Number 0..14dp | NUMERIC(15[1] ,0..14) |
| Short number 0/2dp | NUMERIC(15[1],0/2) |
| Number floating dp | DOUBLE PRECISION |
| **DATE/TIME** |  |
| Short date (all subtypes) | DATE |
| Short time | TIME /TIMETZ\* |
| Datetime (all subtypes) | TIMESTAMP /TIMESTAMPTZ\* |
| **OTHER** |  |
| Boolean | BOOLEAN |
| Sequence | SERIAL/INTEGER [2] |
| Picture | BYTEA |
| List | BYTEA |
| Row | BYTEA |
| Object | BYTEA |
| Binary | BYTEA |
| Item reference | BYTEA |

##### [1] Numeric precision for Number (dp) columns uses the value of $numericprecision. [2] The mapping used for the Omnis Sequence type depends on the value of $sequencetoint. [3] This mapping occurs only if $char38touuid is set to kTrue [4] Use the $addcustomtype() method to add additional mappings, e.g. for XML and JSON \*Time zone data types are used when session.$usetimezone is set to kTrue

#### PostgreSQL to Omnis

|  |  |  |
| --- | --- | --- |
| PostgreSQL Data Type | Description | Omnis Data Type |
| **NUMBER** |  |  |
| INT2/SMALLINT | -32768 to +32767 | Integer 32 bit |
| INT/INT4/INTEGER | -2147483648 to +2147483647 | Integer 32 bit |
| INT8/BIGINT | -2^63 to +2^63-1 | Integer 64 bit |
| SERIAL | 1 to 4294967296 | Integer 32 bit |
| SERIAL8/BIGSERIAL | 1 to 2^64 | Integer 64 bit |
| FLOAT4/FLOAT/REAL | 1E-37 to 1E+37 | Number floating dp |
| DOUBLE/FLOAT8 | 1E-307 to 1E+308 | Number floating dp |
| NUMERIC | Numbers with max precision 1000 | Number floating dp (1) |
| MONEY | -21474836.48 to +21474836.47 | Number 2dp |
| **DATE/TIME** |  |  |
| DATE | Dates only | Short date |
| TIMESTAMP/TIME | Timestamp/time without time zone | Datetime (#FDT) |
| TIMESTAMPTZ/TIMETZ | Timestamp/time with time zone | Character |
| INTERVAL | Flexible format time interval | Character |
| **CHARACTER** |  |  |
| CHAR | Blank-padded characters with size limit | Character |
| VARCHAR | Variable length characters with size limit | Character |
| TEXT | Variable length characters with no size limit | Character |
| JSON/JSONB | JavaScript Object Notation | Character(2) |
| BOOLEAN/BOOL | {'f', 'false', 'n', 'no', '0', 't','true','y','yes','1'} | Boolean |
| CIDR, INET, MACADDR | Strings containing address information | Character |
| UUID | Universally Unique Identifier | Character 36 |
| ENUM | Custom enumerated types | Character 64 |
| XML | Extensible Markup Language content | Character |

|  |  |
| --- | --- |
| OTHERS (including, but not limited to) |  |
| BYTEA, BIT, VARBIT, BOX, CIRCLE, POINT, LINE, PATH, POLYGON, LSEG | Binary |

##### (1) DAM will map decimal values to the Omnis Number dp data type where column scale is <=14 (2) Supported in Studio 8.0.3 and later

### PostgreSQL Troubleshooting

The following points may help in resolving programming issues encountered using PostgreSQL session and statement objects.

1. $rpcparameters(). When calling $rpcparameters(), the DAM uses defaults for the column precision and/or scale since this information is not provided by the pg\_proc system table.  
   For this reason, the API may report parameter-matching problems when calling certain functions and the list (passed to $rpcdefine()) may need to be manually coerced.
2. Error Messages. The following additional error messages may be returned via the session or statement $errortext property:   
   "Native error text could not be retrieved". No connection currently exists to the server or there is no message corresponding to the current error code.

"Unsupported client protocol version". The protocol version reported by the client API is too low. The DAM cannot use this version and you should upgrade to a newer version of the client library. Use the PostgreSQL access library supplied with Omnis Studio.

"Client or interface function not available". The most likely cause of this error is that the client library (or one of its dependencies) was not found and has not been loaded. Can also occur if the client library being used does not provide a required interface function.

"server closed the connection unexpectedly. This probably means the server terminated abnormally before or while processing the request." This error can occur when logging on with a username other than “postgres”. The client library uses the username for the database name unless the database name is specified. Set $database to the required database name (e.g. “postgres”) and try again.

1. Linux Terminal Messages. On Linux, NOTICE and/or WARNING messages are sent to stderr, (normally the terminal window behind Omnis). To avoid these, refer to the server configuration parameter: client\_min\_messages and set his to a higher level.
2. Chunking and Batch Fetching. Chunking of large character/binary data is not handled by DAMPGSQL but is handled automatically by the API. Such data is effectively returned to the DAM as single chunks. $lobthreshold , $lobchunksize and $blobsize therefore have no effect.

Batch fetching of data is also not handled by DAMPGSQL. The API automatically manages transfer of the data and presents the DAM with the entire result set. Hence setting $batchsize has no effect.

Further troubleshooting notes, “how-tos” and tips can be found on the Omnis website at: [www.omnis.net/technotes](http://www.omnis.net/technotes)

## SQLite

This section contains the additional information you need to access a SQLite database, a very popular database which is embedded into a whole range of applications on desktop and mobile devices. The code for SQLite is in the public domain and is thus free for use for any purpose, commercial or private.

SQLite implements a self-contained, server-less, zero-configuration, transactional SQL database engine. Unlike most other SQL databases, SQLite does not have a separate server process. SQLite reads and writes directly to a disk file which can contain multiple tables, indices, triggers, and views. For more information about SQLite and to download it, please go to the website: [www.sqlite.org](http://www.sqlite.org) (portions of this text are taken from the SQLite website).

This section contains the additional information you need to access a SQLite database, including server-specific programming, trouble-shooting and data type mapping to and from the database. For additional information on changes to the SQLite DAM, refer to the readme file which accompanies your installation media.

### Server-specific Programming

#### Logging on to SQLite

To connect using the SQLite DAM, create an object variable of subtype “SQLITESESS”.

You connect to a SQLite data file using the $logon() method. The hostname parameter should be the full path to the data file.

SQLite does not require a username or password, but you can specify a session name that will appear in the SQL Browser and in the Notation Inspector under *$sessions*.

SQLite expects a DOS-style pathname under Windows and an absolute POSIX-style path under macOS and Linux. For example:

###### Do mySession.$logon(‘C:\mydata\mydatafile.db’,’’,’’,’session1’) Returns #F ## on Windows

###### Do mySession.$logon(‘/Users/MyUser/mydatafile.db’,’’,’’,’session1’) Returns #F ## on macOS / Linux

Additionally, you can force SQLite to create the specified data file if it does not exist. To do this, set the $opencreate session property to kTrue before logging on.

To open a read-only connection, set the $readonly session property to kTrue before logging on. (It is not possible to create a data file if the connection is read-only).

If the hostname is ":memory:", then a private, temporary in-memory database is created for the connection. This in-memory database will be deleted when the database connection is closed. Filenames beginning with “:” should be considered reserved for future SQLite extensions and avoided to remove ambiguity. For in-memory databases, $version will be set to “:memory:” following $logon().

If the hostname is an empty string, then a private, temporary on-disk database will be created. This private database will be automatically deleted as soon as the database connection is closed. For temporary databases, $version will be set to “:temporary:” following $logon().

For standard data file connections, $version is read directly from the file header information and reflects the file format version that the data file supports.

#### Transaction Support

SQLite supports both automatic and manual SQL transactions.

To invoke manual transaction mode, the $transactionmode session property should be set to kSessionTranManual.

In this mode you must commence each transaction by calling the $begin() session method and terminating each transaction either by calling $commit(), $rollback(), by switching back to kSessionTranAutomatic or by logging off.

The SQL text that is submitted each time $begin() is called may be augmented using the $transactiontype session property as shown below. The different transaction types affect the way in which SQLite acquires row locks on tables:

|  |  |  |
| --- | --- | --- |
| $transactiontype | Resulting SQL text | Meaning |
| kSQLiteTranDeferred | BEGIN | No locks are acquired on the database until the database is first accessed |
| kSQLiteTranExclusive | BEGIN EXCLUSIVE | EXCLUSIVE locks are acquired on all databases as soon as the BEGIN command is executed |
| kSQLiteTranImmediate | BEGIN IMMEDIATE | RESERVED locks are acquired on all databases as soon as the BEGIN command is executed |

The $commit() and $rollback() methods, invoke the COMMIT and ROLLBACK commands respectively.

$commitmode and $rollbackmode are set to kSessionCommitClose and kSessionRollbackClose respectively for the SQLite DAM. Statement objects are closed upon $commit() / $rollback(). Any pending result set is discarded and the statement is returned to its prepared state ready for re-execution if desired.

#### Incremental BLOB I/O

SQLite supports incremental Input/Output to BLOB columns in database tables. This means that you effectively bind a placeholder for the BLOB at bind time, then write the data to it later. Similarly, you can open a handle to a BLOB which already exists in the database and read/modify its contents without the need to perform a SELECT statement.

To create a placeholder for a BLOB, you should bind the binary variable inside the SQL statement as normal, but set its contents to #NULL. On execution, this creates a *zero-blob* of size *$blobsize-* bytes padded with zeros.

The session object provides several methods for accessing and modifying BLOBs:

1. **$blobopen()**  
   Opens a handle to a BLOB column, identified by its database name, table name, column name and row number, optionally as read-only
2. **$blobclose()**  
   Closes a BLOB handle; you have to close any BLOB handles opened during the session, but any handles left open when the session ends are closed automatically
3. **$blobcloseall()**  
   Closes all BLOB handles
4. **$blobbytes()**  
   Returns the size in bytes that was allocated to a BLOB column when it was created
5. **$blobhandles()**  
   Returns a list of all open BLOB handles including the corresponding database, table name, column name and row number
6. **$blobreopen()**  
   Moves a BLOB handle to a new row within the same table
7. **$blobwrite()**  
   Writes binary data to a BLOB column
8. **$blobread()**  
   Reads binary data from a BLOB column

See the Session Methods section for more details and syntax for these methods.

#### Calculated Columns and Sub-Queries

A column returned from SELECT statement that is the result of an expression or sub-query cannot be described automatically by the DAM because there is no corresponding entry in the sqlite\_master table. In this situation, you can provide a “hint” using a column alias name containing the intended SQL data type, for example:

###### select ’2014-07-27’ as my**date** from table select substr(col1,1,2)||':'||substr(col1,3,2) as id\_**char** from test select 1 as **bool**val, col5+3 as **int**val from test

Other data types recognised include; “timestamp, time, national, tinyint, serial, sequence, dec, float”. When used the emulate OmnisSQL, the DAM also recognises various OmnisSQL function names such as upper(), lower(), ascii(), charindex(), length(), mod(), round(), cos(), sin(), etc. The DAM also recognises literal numeric and integer values, so there is no need to provide alias names for such columns, e.g.

###### select 3.14159, 33\*3, col3+6, sin(0.6), length(col1) from test

Note that there is currently no way to specify the data sub-type when using aliased column names, hence Character data will be *Character 100000000* and numeric data will be *Number floating dp*. If the data type of a calculated column cannot be determined, it will be fetched as binary.

To avoid using column alias names, you can pre-define the fetch list or table instance using a schema, in which case the fetched data will be coersed into the required column types.

### SQLite Encryption

As of Studio 8.0.3, the SQLite DAM supports native datafile *encryption*. When enabled, all data written to the SQLite datafile is encrypted and can only be read and decrypted using the SQLite DAM with the appropriate encryption key.

Encryption is enabled by setting the session object $encryptkey property before logging on to the SQLite datafile. $encryptkey accepts a string of hexadecimal characters. The string should be of even length and should be no longer than 32 characters. The key value will be truncated if it does not meet either of these criteria. The accepted key value is then used to seed an internal private key which is subsequently used by all statement objects belonging to that session object.

To create a new encrypted datafile, the $opencreate property should also be set to kTrue before logging on. For example:

Do sessObj.$opencreate.$assign(kTrue)   ## create a new datafile if it does not exist  
Do sessObj.$encryptkey.$assign(‘1a2b3c4d5e6f’) Returns #F  
Do sessObj.$logon(‘/Users/user1/Desktop/sqlite.db’,’’,’’,’session1’) Returns #F

Once encrypted, $logon() will fail unless the correct $encryptkey is supplied. $encryptkey will be ignored (cleared) if the DAM detects a connection to a non-encrypted datafile. Please note that you cannot change the $encryptkey property while the DAM is logged on. Errors encountered during assignment of $encryptkey are written to session.$nativeerrorcode and session.$nativeerrortext.

The DAM provides two session methods that facilitate encryption/decryption of existing SQLite datafiles:

1. $encrypt(filename)  
   opens a non-encrypted datafile and encrypts it using the $encryptkey. A backup copy of the non-encrypted datafile is created at the file location named *filename.bak*
2. $decrypt(filename)  
   opens a previously encrypted datafile and decrypts it using the $encryptkey. A backup copy of the encrypted datafile is created at the file location named *filename.bak*

$encrypt() and $decrypt() return kTrue on success but will fail, unless the DAM is logged off, if the process cannot get exclusive read/write access to the specified datafile or if *filename.bak*already exists and cannot be overwritten. Once encrypted, connection via third-party tools should be avoided as this may result in undefined behaviour and cause datafile corruption.

##### **Important note**: Your attention is drawn to the terms of the Omnis End User License Agreement and to the following excerpt pertaining to the use of this encryption mechanism and subsequent loss of data and/or of the encryption key(s):

##### Omnis Software disclaims any responsibility for or liability related to the use of this software. IN NO EVENT WILL OMNIS SOFTWARE BE LIABLE FOR ANY INDIRECT, PUNITIVE, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES HOWEVER THEY MAY ARISE AND EVEN IF OMNIS SOFTWARE HAS BEEN PREVIOUSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

### Session Properties

|  |  |
| --- | --- |
| Property | Description |
| $blobsize | The default value for $blobsize is set at 32KB for the SQLite DAM since this property is used routinely when creating empty BLOB columns for use with incremental input/output methods. |
| $encryptkey | Enables encryption. Accepts a string of hexadecimal characters of even length and no longer than 32 chars. The key value is truncated if it does not meet either of these criteria |
| $nullbinary | If kTrue, null-valued binary bind variables are inserted as NULL. If kFalse (the default) they are inserted as zero-blobs of length $blobsize (see Incremental BLOB I/O above). (Studio 8.1.6 and later) |
| $opencreate | If kTrue, the data file specified at $logon() will be created if it does not exist; in this case, the datafile is encrypted. If kFalse (the default), an error will be generated if the data file is not found. |
| $readonly | If kTrue, the connection will be opened in read-only mode. All attempts to write to the data file will fail with an error. If kFalse (the default), read and write operations are permitted. |
| $transactiontype | Specifies the locking behavior for manual transactions. This is one of the following constants:  kSQLiteTranImmediate,  kSQLiteTranExclusive or  kSQLiteTranDeferred (the default). |

### Session Methods

|  |  |
| --- | --- |
| Method | Description |
| $blobbytes() | $blobbytes(iBlobHandle) returns the size in bytes that was allocated to a BLOB column when it was created |
| $blobclose() | $blobclose(iBlobHandle) closes a BLOB handle. You should close any BLOB handles opened during the session. Any handles left open when the session ends are closed automatically however. Always returns kTrue |
| $blobcloseall() | $blobcloseall() closes all BLOB handles. This method always returns kTrue |
| $blobhandles() | $blobhandles(lHandleList ) returns a list of all BLOB handles including their corresponding database, table names, column names and row numbers. Aborted/invalid handles are shown with a row number set to zero. Returns kTrue on success, otherwise kFalse |
| $blobopen() | $blobopen(cDatabase, cTable, cColumn, iRow [,bReadOnly]) opens a handle to a BLOB column, identified by its database name, table name, column name and row number, optionally as read-only. Returns a BLOB handle on success or zero on failure. The SQLite DAM numbers BLOB handles incrementally starting from 1001 |
| $blobread() | $blobread(iBlobHandle, xBinary [,iSize ,iOffset]) reads binary data from a BLOB column into the supplied binary variable. If iSize is omitted, the value of $blobsize is assumed |
| $blobreopen() | $blobreopen(iBlobHandle, iRow) moves a BLOB handle to a new row within the same table. If iRow exceeds the number of rows in the table, this invalidates the handle. Only the row number can be modified. To change the database, table name or column name, a new handle should be opened |
| $blobwrite() | $blobwrite() writes binary data to a BLOB column |
| $encrypt() | $encrypt(cFilename) opens a non-encrypted datafile and encrypts it using the $encryptkey. A backup copy of the non-encrypted datafile is created at the file location named *filename.bak* |
| $decrypt() | $decrypt(cFilename) opens a previously encrypted datafile and decrypts it using the $encryptkey. A backup copy of the encrypted datafile is created at the file location named *filename.bak* |
| $insert\_list() | Studio 9 and later. $insert\_list(cTable, lListData) inserts each row from lListData into cTable. The database must be logged-on and the table must already exist. It is also assumed that the list definition is compatible with the table definition. |
| $lastrowid() | $lastrowid() returns the rowid of the most recent successful INSERT into the database from the current connection |
| $rowsmodified() | $rowsmodified() returns the total number of database table rows that have been affected by INSERT, UPDATE and DELETE operations since the connection was opened (includes all statement objects) |

### Data Type Mapping

#### Omnis to SQLite

The SQLite DAM creates custom data types in order to preserve information about Omnis subtypes, notably: DATE(n) as well as PICTURE, LIST, ROW, OBJECT and OBJECTREF. There are also single mappings for Omnis Character and National data, implying that CHAR(n) and NCHAR(n) can store up to the maximum field length supported by Omnis (10000000 characters). This is contrary to other relational databases which impose a fixed size on such columns.

Although this greatly improves compatibility between Omnis and SQLite, if portability of the data file is of concern, then it may be preferable to avoid using $createnames() / $coltext() in favor of manual statements that use standard SQL types, e.g. VARCHAR(n), DATE, TEXT and BLOB.

|  |  |
| --- | --- |
| Omnis Data Type | SQLite Data Type |
| **CHARACTER** |  |
| Character n | CHAR(n) |
| National n | NCHAR(n) |
| **NUMBER** |  |
| Integer 64 bit | BIGINT |
| Integer 32 bit | INTEGER |
| Short integer | TINYINT UNSIGNED |
| Number 0..14dp | NUMERIC(15, 0..14) |
| Short number 0/2dp | NUMERIC(9, 0/2) |
| Number floating dp | FLOAT |
| **DATE/TIME** |  |
| Short date 1900..1999  Short date 1980..2079  Short date 2000..2099 | DATE(1900)  DATE(1980)  DATE(2000) |
| Short time | TIME |
| Datetime (all subtypes) | TIMESTAMP |
| **OTHER** |  |
| Boolean | BIT |
| Sequence | INTEGER PRIMARY KEY (auto increments when inserted as NULL) |
| Picture | PICTURE |
| List | LIST |
| Row | ROW |
| Object | OBJECT |
| Object reference | OBJECTREF |
| Binary / other | BINARY |

#### SQLite to Omnis

The SQLite DAM recognises several additional SQL data types in order to maximise compatibility with externally generated data files as well as those generated by Omnis.

|  |  |
| --- | --- |
| SQLite Data Type | Omnis Data Type |
| **NUMBER** |  |
| TINYINT UNSIGNED | Short integer 0..255 |
| TINYINT, INT, SMALLINT, INTEGER | Integer 32 bit |
| SEQUENCE, INT AUTO INCREMENT | Sequence |
| BIGINT | Integer 64 bit |
| FLOAT, REAL, DOUBLE | Number floating dp |
| NUMERIC(p,s), DEC(p,s), DECIMAL (p,s) | Short number s dp (p <=9, s=0 or 2) Number s dp (p <= 15)  Number floating dp (p > 15) |
| **DATE/TIME** |  |
| DATE(1900)  DATE(1980)  DATE(2000) | Short date 1900..1999  Short date 1980..2079  Short date 2000..2099 |
| DATE, TIMESTAMP, TIME | Date Time (#FDT) |
| **CHARACTER** |  |
| CHAR, VARCHAR, TEXT, CLOB, | Character |
| NCHAR, NVARCHAR, NATIONAL | National |
| **OTHERS** |  |
| BOOLEAN, BOOL, BIT | Boolean |
| PICTURE | Picture |
| LIST | List |
| ROW | Row |
| OBJECT | Object |
| OBJECTREF | Object reference |
| BINARY / other | Binary |

### Troubleshooting

The following points may help in resolving programming issues encountered using SQLite session and statement objects.

For additional updated troubleshooting issues, refer to the readme file which accompanies the installation media.

For a detailed explanation of the SQL syntax supported by SQLite, please refer the SQLite website: [www.sqlite.org](http://www.sqlite.org)

1. SQLite does not currently support dynamic creation of SQL stored procedures or functions. The associated methods; $rpcprocedures(), $rpcparameters() & $rpc() therefore return kFalse.
2. The SQLite API handles the transfer of binary data automatically. The $blobsize, $lobchunksize and $lobthreshold properties are therefore ignored.
3. For performance reasons, journaling mode is set to PERSIST for the SQLite DAM. For optimum performance, especially on Linux it may be desirable to turn off journaling, (“PRAGMA journal\_mode = OFF”). **Note:** in this mode however it will not be possible to rollback manual transactions.
4. You may experience slow performance during certain INSERT operations. Each INSERT and UPDATE operation is normally committed to the disk drive so as to preserve integrity of the data in the event of a crash or power failure. Executing "PRAGMA synchronous=OFF" tells SQLite not to wait for data to reach the disk surface between writes which results in much faster performance. This risks data loss or corruption in the event of a crash however. Alternatively, you can use manual transaction mode (kSessionTranManual) to commit several INSERT operations at once.

* A column returned from SELECT statement that is the result of an expression or sub-query may require an additional column name alias containing the intended data type. Refer to *Calculated Columns and Sub-Queries* above for further details.

## ODBC

This section contains the additional information you need to access a database using ODBC middleware, including server-specific programming, data type mapping, as well as troubleshooting. For general information about logging on and managing your database using the Omnis SQL Browser, refer to the earlier parts of this manual.

### Properties and Methods

In addition to the “base” properties and methods documented in the *SQL Programming* chapter, the ODBC DAM provides the following additional features.

#### Session Properties

|  |  |
| --- | --- |
| Property | Meaning |
| $dbmsname | Once a session has been established this is the name of the database that the object is connected to. This defaults after a $logoff. (Read only) |
| $dbmsversion | Once a session has been established this is the version of the database that the object is connected to. This defaults after a $logoff. (Read only) |
| $defaultdatabase | When set, the session will attempt to log on to the database specified. A change to $defaultdatabase must be made before logging on, otherwise the change will not take effect until the session is re-used. To stop using a default database for the session, set $defaultdatabase to an empty string (the default value). This property may not be supported by all DBMS vendors. |
| $drivername | Once a session has been established this is the name of the ODBC driver that the object is using. This defaults after a $logoff. (Read only) |
| $driverversion | Once a session has been established this is the version of the ODBC driver that the object is using. This defaults after a $logoff. (Read only) |
| $driverodbcversion | Once a session has been established this is the version of the ODBC API that the driver supports. This defaults after a $logoff. (Read only) |
| $infoaserror | If kTrue (the default), execution results that report SQL\_SUCCESS\_WITH\_INFO are reported as errors. If kFalse, the DAM treats this the same as SQL\_SUCCESS and ignores the accompanying message. Studio 5.2 and later. |
| $logontimeout | The timeout in seconds for a $logon() call. The default is 15 seconds. A value of 0 represents no timeout. A value of –1 can also be specified to indicate that the DAM should not attempt to set a timeout value. |
| $mode | macOS and Linux only. A *kODBCMode…*value used to select the ODBC driver manager library for non-standard ODBC connections, e.g. SAP SQL Anywhere. |
| $programname | The name to be registered at the server for the process associated with the session. By default, $programname is set to the current library name. This property may not be supported by all DBMS vendors: see the $useprogramname property. |
| $querytimeout | The timeout in seconds for a query. A value of 0 represents no timeout, which is the default. |
| $savefile | Used in conjunction with $usefiledsn. If kTrue, invokes the SAVEFILE ODBC connection attribute which writes updated connection details back to the specified File DSN. |
| $timezone | The local timezone offset relative to GMT. The initial value is read from the OS when the session object is created and will be of the form “+/-HH:MM”. See $usetimezone. |
| $trustedconnection | Supported values are kODBCIgnoreTrusted (the default,) kODBCUseTrusted and kODBCNotTrusted. When a value of kODBCUseTrusted is specified, the session attempts to log on to the DBMS using a server trusted connection, for which the $username and $password will be ignored. When a value of kODBCNotTrusted is specified, the session attempts to log on to the DBMS with an explicitly non-trusted connection. This property may not be supported by all DBMS vendors.  Note that to enforce trusted connections, it may be necessary to disable server prompting by setting $uselogonprompt to kFalse. |
| $usefiledsn | If kTrue, the hostname specified at logon will be treated as a file DSN. The default is kFalse. Not all drivers support the use of file DSNs. |
| $uselogonprompt | Governs the use of logon prompts where there is insufficient information to connect Can also used to force the ODBC Administrator library to display a configuration dialogue when connecting to File DSNs. $uselogonprompt accepts constant values of: kODBCPromptNever (0), kODBCPromptComplete (1), kODBCPromptAlways (2). And kODBCPromptDsnLess (3). Not all drivers can support this feature. |
| $useprogramname | If kTrue, the session attempts to register $programname as the process name when logging on to the server. This property may not be supported by all DBMS vendors. Hence the default value for $useprogramname is kFalse. For SQLServer, the program name can be found in the sysprocesses table of the master database. |
| $usequalifiers | When set to kTrue, the DAM treats qualified table names as owner.tablename. To prevent this, for instance when using a text file driver, set $usequalifiers to kFalse. This property affects the behaviour of the $columns(), $tables(), $indexes() & $results() session methods. $usequalifiers is ignored until the session logs on, at which time the default value is determined and $usequalifiers is overwritten. |
| $usescale | If kTrue, Omnis number dp columns are bound using a precision of 15 + the dp value. If kFalse (the default), number dp columns are bound using a precision of 15. Also affects $createnames(). Studio 8.1.5 and later. |
| $usetimezone | If kTrue, $timezone will be applied to values inserted and fetched from TIME & TIMESTAMP columns, that is; datetime values are subject to modification by comparing the local timezone with the server’s timezone. Applies to MS-SQL Server connections only. |
| $usevarcharmax | If kTrue, Character columns > 4000 map to VARCHAR(MAX) / NVARCHAR(MAX) dependent on the value of $unicode. If kFalse, they map to TEXT / NTEXT. Assumes the connection is to MS SQL Server. |
| $nationaltowchar | Available only with the Unicode DAM. By default, Omnis Character and National fields are mapped to the SQL\_WCHAR, SQL\_WVARCHAR and SQL\_WLONGVARCHAR data types. By setting $nationaltowchar to kTrue only National fields will be mapped to these types (to the equivalent server data types) and Character fields will be mapped to SQL\_CHAR, SQL\_VARCHAR and SQL\_LONGVARCHAR as determined by the Omnis field length. Character fields mapped in this way are subject to data loss/truncation where such fields contain Unicode characters. When setting this property, please note that Unicode data types usually have precision limits half that of their corresponding ANSI data types. For example, this is 8000 for the SQL Server VARCHAR() data type but 4000 for NVARCHAR(). $nationaltowchar affects both the text returned by the $createnames() method and the binding of input parameters. |
| $datesecdp | The $datesecdp property specifies the number of decimal places used for server date columns. For use with Microsoft SQL Server 2008 TIME and DATETIMEOFFSET data types which include a scale parameter. $datesecdp affects the string returned by $createnames() as well as input binding. Defaults to 2 but valid values are in the range 0 to 7. The property is set to zero for a MyODBC connection to allow correct type mapping to DATETIME. |
| $defaultschema | For use with Microsoft SQL Server 2005 and later. $defaultschema returns the schema name which owns tables created by the current user. This should be used in place of username in methods such as $tables(). Assigning to this property invokes an ALTER USER statement which changes the default schema for the user. |

A number of additional session properties have been added to the ODBC DAM in Studio 4.3.1 to facilitate better understanding and control of cursors and transactions. Use of these properties assumes that the session is logged-on and has been placed in manual transaction mode (kSessionTranManual):

|  |  |
| --- | --- |
| Property | Description |
| $autobegintran | (Read Only). This property always returns kTrue for DAMODBC because the ODBC API implicitly starts transactions, even in manual transaction mode. |
| $cursorsensitivity | (Read Only). This property returns kTrue if SQL cursors are sensitive to changes made by other cursors within the same transaction. kFalse is returned if results returned by cursors are not sensitive to changes made by other cursors in the same transaction. |
| $txncapability | (Read Only). Returns one of the constant values listed in the Catalog under ODBCDAM-Transaction Types. Certain drivers only support use of DML statements within transaction blocks (SELECT, INSERT, UPDATE, DELETE). Others may ignore or permit use of DDL statements (CREATE TABLE, DROP INDEX, and so on) but may require the transaction block to be committed immediately. kODBCTxnAll specifies that transactions can contain DDL statements and DML statements in any order. |
| $multipletransactions | (Read Only). Returns kTrue if the driver supports more than one active transaction at the same time, kFalse if only one transaction can be active at any time. |
| $multipleresultsets | (Read Only). Returns kTrue if the data source supports multiple result sets, kFalse if it does not. |
| $isolationoptions | (Read Only). Returns a bitmask value representing the transaction isolation levels supported by the driver. The bit positions correspond to the constant values listed in the Catalog under ODBCDAM-Isolation Levels. |
| $isolationlevel | Returns the current transaction isolation level in use by the session. To change the isolation level, assign one of the constants listed in the Catalog under ODBCDAM-Isolation Levels. The isolation level must be one the levels advertised by $isolationoptions.  Changing the isolation level implicitly invokes a $commit(). |

#### Session Methods

|  |  |
| --- | --- |
| Method | Description |
| $getdrivers() | SessObj.$getdrivers(lResult) retrieves a list of all ODBC drivers installed on the system. lResult is populated with the list of drivers installed and is defined with the following character columns:  DriverName The alternate driver name (i.e. the descriptive name) Version The version string reported by the driver CompanyName The Company name embedded within the driver file FileName The physical path and file name of the driver  CompanyName is only obtainable for Win32 and will return as empty for other platforms. DriverName is obtainable directly from the driver only for Win32. Other platforms require each driver to be loaded and called in order to obtain the version string. Hence, there will be a commensurate delay when calling this method.  Example: Do sessObj.$getdrivers(iDriverList) Returns #F |
| $getdatasources() | SessObj.$getdatasources(lResult, kDSNMode) retrieves a list of ODBC DSNs of type specified by kDSNMode which should be passed as either kODBCSystemDSN or kODBCUserDSN. $getdatasources() does not support File DSNs (see below). On return, lResult is defined with two character columns:  DSNName The User assigned name for the data source Driver The alternate name of the driver associated with the data source  Example: Do sessObj.$getdatasources(iDSNList,kODBCSystemDSN) Returns #F |
| $getinfo() | $getinfo(lResult, cDSNName, kDSNMode) retrieves the information defined for the specified data source or driver as a list of keyword-value pairs. kDSNMode should be passed as either kODBCSystemDSN, kODBCUserDSN or kODBCDriverInfo.  $getinfo() does not support File DSNs for which standard FileOps methods can be used to read/modify as required. On return lResult is defined with the following character columns:  KeyWord The name of the DSN/driver attribute Value The value of the DSN/driver attribute  Example: Do sessObj.$getinfo(iDSNinfo,’myDsn’,kODBCUserDSN) Returns #F |
| $setinfo() | $setinfo(cDSNName, kDSNMode, lData) writes the information contained in lData to the specified Data source or Driver key in the system information. lData should be defined with Keyword and Value columns as returned by $getinfo(). If kDSNMode is kODBCDriverInfo, this has the effect of modifying system information for the specified driver. cDSNName should contain the descriptive name of the ODBC Driver as opposed to the physical file name.  If kDSNMode is kODBCSystemDSN or kODBCUserDSN, this has the effect of modifying the specified data source. $setinfo() does not register a new data source or driver although it will write data to the DSN as though it already exists. To properly create a data source; use the $configdsn() method instead. To properly register a driver, you should refer to the vendor’s installation program.  Example: Do sessObj.$setinfo(‘myDsn’,kODBCUserDSN,iDSNinfo) Returns #F |
| $configdsn() | $configdsn(kDSNMode, kRequestType, cDriverName, lAttributes) allows the specified datasource to be created, modified or removed. kDSNMode should be either kODBCSystemDSN or kODBCUserDSN. $configdsn() does not support configuration of File DSNs- for which an alternative strategy is provided. kRequestType should be passed as either kODBCAddDSN, kODBCModifyDSN or kODBCRemoveDSN. cDriverName should correspond with the descriptive name of the driver (i.e. not the physical file name). lAttributes should be defined with two character columns and is used to pass keyword-value pairs to the driver manager sufficient to perform the required action. Usually this involves adding a single line to the list to identify the DSN to be created/modified/removed, e.g.  KeyWord Value  DSN dsnname  but can also include other keywords that are allowed by the driver. When $uselogonprompt is set to kODBCPromptNever, this prevents $configdsn() from opening setup dialogues. The DSN is created/modified silently using values read from the attribute list instead.  Example: Do sessObj.$configdsn(kODBCUserDSN,kODBCAddDSN, ’SQL Server’,lAttribList) Returns #F |
| $getoption() | $getoption(kOption, cAttribute) allows the value of an ODBC configuration attribute to be retrieved. kOption should be passed as one of the following constants:  kODBCTrace Requests the TRACE on/off flag kODBCTraceLib Requests the name and path to the ODBC trace library kODBCTraceFile Requests the name and path to the ODBC trace log kODBCFileDSNDir Requests the default directory containing file DSNs kODBCPerfMon Requests the Performance monitoring on/off flag kODBCRetryWait Requests the connection pool RetryWait timeout  On return, cAttribute contains the value of the requested option as a character string.  Example: Do sessObj.$getoption(kODBCFileDSNDir,iFileDSNDir) Returns #F |
| $setoption() | $setoption(kOption, cAttribute) allows the value of an ODBC configuration attribute to be modified. kOption should be either kODBCTrace, kODBCTraceLib, kODBCTraceFile, kODBCFileDSNDir, kODBCPerfMon or kODBCRetryWait. cAttribute should contain a character string representing the new value for the specified configuration option.  Example: Do sessObj.$setoption(kODBCTraceFile,iTraceFile) Returns #F |

### Connecting to your Database

To connect to your database using ODBC, you need to create a session object with a subtype of ODBCSESS. In order to log on to the database using the SessObj.$logon() method, the hostname must contain the data source name from the ODBC control panel and the user name and password should contain the values required by the data source. If using a SQL Server data source that uses trusted connections, the user name and password are omitted, for example

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

When the session property $usefiledsn is set to kTrue, this specifies that the hostname parameter is to be treated as a file data source name by the driver manager.

When the session property $uselogonprompt is set to kODBCPromptComplete, this specifies that the driver will prompt for missing logon information. Note that not all drivers support prompting and this may result in the logon failing.

#### Making a DSN-less Connection

To make a connection without using an ODBC DSN, all of the information necessary to make a connection needs to be passed as an *ODBC connection string* and the session $uselogonprompt property needs to set to kODBCPromptDsnLess.

The connection string is passed to the $logon() method via the hostname parameter. The username and password parameters are left blank.

ODBC connection strings consist of keyword-value pairs separated by semi-colons and are database specific. Examples of such strings include:

For SQL Server:

###### Driver=SQL Server; Server=192.168.0.10; Database=accounts; Uid=fred; Pwd=secret

For the Omnis ODBC Driver:

###### Driver=Omnis ODBC Driver; DataFilePath=c:\TRAVEL.DF1; Username=myuser; Password=mypassword

For specific details on connection strings, please refer to the documentation supplied with the RDBMS or with the ODBC driver.

### Transactions

Generally, using manual transaction mode results in increased performance because the session object does not force a commit after each statement.

If you do not have a result set pending, ODBC session objects will commit each statement if the transaction mode is automatic. If the transaction mode is server, the session may be committed depending on the behavior of the ODBC driver.

### Dates

The session property $defaultdate allows default values to be added to date values mapped to the server where the Omnis date value does not contain complete information, e.g. a Short time mapped to a server date time.

### Multiple cursors

To allow multiple select cursors when connecting to Microsoft SQLServer the statement issuing the SELECT must have the $usecursor property set to kTrue before the statement is executed. If a statement is issued when $usecursor is kFalse and this statement returns a result set, this will prevent all other statements in the same session from returning data. The blocking results must be completely processed or cleared before another result set can be generated. If a commit or rollback is performed on the session, all the session’s statement cursors will be closed and all pending results will be lost. Note that a SQLServer cursor only allows SQL SELECT and EXECUTE procedure commands to be issued.

### SQL Server 2000 Data Types

The following new types were introduced with Microsoft SQLServer2000:

SQL\_BIGINT

Values fetched into Omnis from BIGINT columns are converted into the Character 20 type. This is necessary since BIGINTs are stored in 64 bits, giving them a range of ±263 or

-9223372036854775808 to 9223372036854775807. The largest numeric value which can be stored using the Integer 32 bit type is ±231 or –2147483648 to 2147483647.

Omnis Character variables can be input into BIGINT columns provided that the character length (precision) does not exceed 19.

Note: Hash variables such as #S1 cannot be bound as input variables for BIGINT columns since their length is preset to 10,000,000.

**SQL\_VARIANT**

Values fetched from SQL\_VARIANT columns are converted into the Binary type so they can be preserved in their raw format.

Since the data type, precision and scale are not known prior to fetching, it may be necessary to pre-process the table before retrieving the variant data. This is done using the SQL\_VARIANT\_PROPERTY() function to build a list of variant types contained in a specified column.

###### Do myStatement.$execdirect(‘select cast(SQL\_VARIANT\_PROPERTY(col3,'BaseType') as char(32)) from mytable’) returns #F

When fetching the data into Omnis, the CAST() function can then be used inside the SELECT statement to ensure that the incoming data gets converted to the proper Omnis types.

###### Do myStatement.$execdirect(‘select CAST(col3 as smalldatetime) from mytable’) returns #F

For this reason, tables containing SQL\_VARIANTs are probably best used in tandem with an index column, which is used to associate a cast type with each row. Since there is no variant type in Omnis, there is no systematic way of reading a whole column of variants.

Note: text, ntext, image & timestamp types are not supported by SQL\_VARIANT.

**Custom Data Types**

When custom data types are fetched from SQL Server their base type is abstracted from the custom type and returned to Omnis. For example, if a custom data type is created using;

###### Do myStatement.$execdirect(‘EXEC sp\_addtype birthday, datetime, 'NOT NULL') returns #F Do myStatement.$execdirect(‘create table test(col1 integer,col2 birthday)’) returns #F

Then, a $columns() performed on the table, describes col2 as DATETIME

If a custom data type is specified when creating or altering a table, this is passed straight through the DAM.

**TABLE**

The TABLE data type can be used in two ways.

The first is as a local variable in a user SQL function. Local variables are defined as type **table** and can be used to temporarily store the result of a query.This usage is beyond the scope of Omnis however.

The second is as the **return value** from a user-defined function. For example, the following function defines a table as it’s return value. (The table must be defined in the RETURNS section.):

###### CREATE FUNCTION Function1 ( @Param integer ) RETURNS @myTable TABLE   (     col1 integer, col2 char(32)   ) AS BEGIN   INSERT @myTable **SELECT \*** FROM valuetest WHERE col1 > @Param   RETURN END

Omnis can then call the function to obtain the table results, e.g.:

###### Do mylist.$define(col1,col2) Do myStatement.$execdirect(‘select \* from Function1(50)’) returns #F Do myStatement.$fetch(myList,kFetchAll)

### ODBC Administration

The $getdrivers(), $getdatasources(), $getinfo(), $setinfo(), $configdsn(), $getoption() and $setoption() session methods (documented above) allow the ODBC DAM to be used to add, modify and remove ODBC Data Source Names (DSNs) as well as to retrieve and modify information about ODBC drivers and general ODBC administration attributes.

The $uselogonprompt property has been modified to allow driver prompting to be forced if required.

All of these methods return a boolean value to indicate successful operation. Errors generated by these methods are returned as normal via the session object’s $nativeerrorcode and $nativeerrortext properties.

#### Configuration of File DSNs

Using $getoption() to retrieve the default directory for file DSNs allows the FileOps component to be used together with other 4GL techniques in configuring File datasources.

To create a File DSN, you should prompt for the new filename (the DSN Name) and use FileOps to create the new file. One or more KeyWord-value pairs should also be written to the file, e.g. DRIVER=drivername- the minimum requirement for a File DSN. To complete the setup of the new DSN, you should follow the procedure for modifying/testing the File DSN.

To modify or test a File DSN, use the following procedure:

1. Set $usefiledsn to kTrue
2. Set $savefile to kTrue
3. Set $uselogonprompt to kODBCPromptAlways
4. Execute $logon() with the name of the File DSN as the hostname.

Under Win32, this prompts the ODBC Administrator library to display the driver specific logon dialogue which prompts for information necessary to make the connection. If the DAM is successful in logging on (and $savefile is set to kTrue), the Administrator library writes the additional information back to the File DSN, hence modifying the datasource.

To remove a File DSN, you should use the FileOps component to manually delete the specified filename.

#### macOS and Linux Considerations

Under Unix, the DAM locates user DSN information (odbc.ini) using the value of the ODBCINI environment variable if is set. If ODBCINI is not set, the DAM attempts to use the ".odbc.ini" (hidden file) in your user’s home directory or failing that, it defaults to "/Library/ODBC/odbc.ini".

The DAM locates system DSN information using the value of the ODBCSYSINI environment variable if it is set. If ODBCSYSINI is not set, the DAM attempts to locate the system driver information using the value of ODBCINSTINI instead. If ODBCSYSINI is set, this location is also assumed for the location of the driver information file (odbcinst.ini). Note that if set, ODBCSYSINI should identify a folder only- not a file name.

The ODBC Driver manager used on your system must support the necessary API calls in order to perform certain administration functions (editing and modifying DSN information, for example). If the required API calls are missing, these functions will not be available.

### ODBC Troubleshooting

The following points may help in resolving issues in programming applications that use ODBC session objects.

1. ODBC does not support any extended ORACLE cursor operations such as positioned update and delete.
2. You must specify literals in SQL statements with single quotes ('), not double quotes (").
3. Some data sources may strip trailing spaces prior to sending it to the session object. SQL Server behaves in this way.

Further troubleshooting notes, “how-tos” and tips can be found on the Omnis website at: [www.omnis.net/technotes](http://www.omnis.net/technotes)

### ODBC Data Type Mapping

The following table describes the data type mapping for Omnis to ODBC connections.

The Omnis to ODBC mapping will attempt to pick the best match based on the types the driver supports in the order listed. For example, if the driver supports SQL\_VARCHAR and SQL\_CHAR data up to a maximum column size of 255, but SQL\_LONGVARCHAR data up to 2 GB, an Omnis Character(1000) will map to whatever the associated server native type is for SQL\_LONGVARCHAR, e.g. TEXT.

#### Omnis to ODBC

|  |  |
| --- | --- |
| Omnis Data Type | ODBC Data Type |
| **CHARACTER** |  |
| Character(n) National(n) | [1]SQL\_VARCHAR(n)  [1]SQL\_CHAR(n)  [1]SQL\_LONGVARCHAR(n)  SQL\_CLOB(n) (DB2 only) |
| **DATE/TIME** |  |
| Short date (all subtypes) | SQL\_DATE SQL\_TYPE\_DATE SQL\_TIMESTAMP SQL\_TYPE\_TIMESTAMP |
| Short time | SQL\_TIME SQL\_TYPE\_TIME SQL\_TIMESTAMP SQL\_TYPE\_TIMESTAMP |
| Date time (#FDT) | SQL\_TIMESTAMP SQL\_TYPE\_TIMESTAMP |
| **NUMBER** |  |
| Short integer (0 to 255) | SQL\_TINYINT (unsigned) SQL\_SMALLINT |
| Integer 64 bit | SQL\_BIGINT  SQL\_CHAR(20) |
| Integer 32 bit  Sequence | SQL\_INTEGER  SQL\_NUMERIC(10,0) SQL\_DECIMAL(10,0) SQL\_FLOAT SQL\_DOUBLE |
| Short number 0-2dp Number floating dp, 0..14 dp | [2]SQL\_NUMERIC(p,s)  [2]SQL\_DECIMAL(p,s)  SQL\_FLOAT SQL\_DOUBLE |
| **OTHER** |  |
| Boolean | SQL\_BIT SQL\_TINYINT SQL\_SMALLINT SQL\_NUMERIC(1,0) SQL\_DECIMAL(1,0) SQL\_CHAR(1) SQL\_VARCHAR(1) SQL\_FLOAT |
| Picture, Binary, List, Row, Object, Item reference | SQL\_VARBINARY(blobsize) SQL\_BINARY(blobsize) SQL\_LONGVARBINARY(blobsize) SQL\_BLOB(blobsize) (DB2 only) Where blobsize is SessObj.$blobsize |

##### [1] Refer to the $nationaltowchar property for use with the Unicode version of Omnis Studio [2] As of Studio 8.1.5, $usescale can be used to calculate p as 15 + s. E.g. In this mode a Number 14dp will map to NUMERIC(29,14) giving 15 scalar digits plus 14 mantissa digits.

#### ODBC to Omnis

|  |  |
| --- | --- |
| ODBC Data Type | Omnis Data Type |
| **CHARACTER** |  |
| SQL\_CHAR(n) SQL\_VARCHAR(n) SQL\_LONGVARCHAR(n) SQL\_WCHAR(n) SQL\_WVARCHAR(n) SQL\_WLONGVARCHAR(n) SQL\_CLOB(n) | Character(n) |
| **DATE/TIME** |  |
| SQL\_DATE SQL\_TYPE\_DATE | Short date 1980 |
| SQL\_TIME SQL\_TYPE\_TIME | Short time |
| SQL\_TIMESTAMP SQL\_TYPE\_TIMESTAMP | Date time (#FDT) |
| **NUMBER** |  |
| SQL\_DECIMAL(p,s) SQL\_NUMERIC(p,s) | Number (s)dp |
| SQL\_SMALLINT | Integer 32 bit |
| SQL\_TINYINT (unsigned) | Short integer |
| SQL\_TINYINT (signed) | Integer 32 bit |
| SQL\_INTEGER | Integer 32 bit |
| SQL\_BIGINT | Integer 64 bit |
| SQL\_REAL SQL\_FLOAT SQL\_DOUBLE | Number floating dp |
| SQL\_BIGINT | Character 20 |
| **OTHER** |  |
| SQL\_BIT | Boolean |
| SQL\_BINARY SQL\_VARBINARY SQL\_LONGVARBINARY SQL\_BLOB SQL\_GUID | Binary |
| SQL\_VARIANT | Binary |
| Custom Data Types | N/A |
| Table Type | N/A |

## JDBC

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

This section contains the information you need to access a database using the JDBC object DAM and JDBC drivers (plus their associated middleware where applicable), including server-specific programming, data type mapping and troubleshooting. For general information about logging on and managing your database using the SQL Browser, refer to the earlier parts of this manual.

### Minimum Requirements

To use the JDBCSESS object the client machine must have the Java Runtime Environment v1.4 or higher installed. In addition, the JDBCSESS object will only support JDBC 2.x certified drivers. The JDBC DAM utilises the Omnis Java Engine (OJE), so it is important that the requirements for the OJE are also met. In particular a JVM\_PATH environment variable must be set to the path of the JVM library in order for the OJE to start the Java Virtual Machine.

### Properties and Methods

In addition to the “base” properties and methods documented in the *SQL Programming* chapter, the ODBC DAM provides the following additional features.

#### Session Properties

|  |  |
| --- | --- |
| Property | Meaning |
| $dbmsname | Once a session has been established this is the type of database that the object is connected to. This defaults after a $logoff. (Read only) |
| $dbmsversion | Once a session has been established this is the version of the database software that the object is connected to. This defaults after a $logoff. (Read only) |
| $drivername | Prior to a session being established this should be set to the name of the JDBC driver that the object wishes to use in order to establish a connection. This can also be set using the $setdriver() method. |
| $driverversion | Once a session has been established this is the version of the JDBC driver that the object is connected to. This defaults after a $logoff. (Read only) |
| $logontimeout | The timeout in seconds for a $logon call. The default is 15 seconds. A value of 0 represents no timeout. |

#### Session Methods

|  |  |
| --- | --- |
| Property | Meaning |
| $setdriver() | $setdriver(‘drivername’) sets the JDBC driver that the session object should use to establish a connection. This is the same as assigning a name to $drivername. |

### Connecting to your Database

To connect to your database, you need to create a session object with a subtype of JDBCSESS. Unlike the other Session objects, JDBC requires a pre-logon step of specifying the JDBC driver name to be used. This is performed using the session method $setdriver(). For example:

###### Do SessObj.$setdriver(‘sun.jdbc.odbc.JdbcOdbcDriver’)

Failure to perform this step will cause the $logon() to fail. In order for the specified JDBC driver to be successfully loaded, it must exist in the system CLASSPATH environment variable.

To log on to the database using the SessObj.$logon() method, the hostname must contain the database URL required by the specified driver. The user name and password should contain the values required by the database.

### Transactions

Generally, using manual transaction mode results in increased performance because the session object does not force a commit after each statement.

If you do not have a result set pending, JDBC session objects will commit each statement if the transaction mode is automatic. If the transaction mode is server, the session may be committed depending on the behavior of the JDBC driver.

### Dates

The session property $defaultdate allows default values to be added to date values mapped to the server where the Omnis date value does not contain complete information, e.g. a Short time mapped to a server date time.

### Multiple cursors

To allow multiple select cursors when connecting to Microsoft SQLServer, the statement issuing the SELECT must have the $usecursor property set to kTrue before the statement is executed. If a statement is issued when $usecursor is kFalse and this statement returns a result set, this will prevent all other statements in the same session from returning data. The blocking results must be completely processed or cleared before another result set can be generated. If a commit or rollback is performed on the session, all the session’s statement cursors will be closed and all pending results will be lost.

### JDBC Data Type Mapping

The following table describes the data type mapping for Omnis to JDBC connections. The Omnis to JDBC mapping will attempt to pick the best match based on the types the driver supports in the order listed. For example, if the driver supports VARCHAR and CHAR data up to a maximum column size of 255, but LONGVARCHAR data up to 2 gig, an Omnis Character(1000) will map to whatever the associated server native type is for LONGVARCHAR, e.g. TEXT.

#### Omnis to JDBC

|  |  |
| --- | --- |
| Omnis Data Type | JDBC Data Type |
| **CHARACTER** |  |
| Character(n) National(n) | VARCHAR(n) CHAR(n) LONGVARCHAR(n)  CLOB(n) |
| **DATE/TIME** |  |
| Short date (all subtypes) | DATE  TIMESTAMP |
| Short time | TIME  TIMESTAMP |
| Date time (#FDT) | TIMESTAMP |
| **NUMBER** |  |
| Short integer (0 to 255) | SMALLINT |
| Sequence Integer 32 bit | INTEGER NUMERIC(10,0) DECIMAL(10,0) FLOAT DOUBLE |
| Integer 64 bit | BIGINT |
| Short number 0-2dp Number floating dp, 0..14 dp | FLOAT DOUBLE |
| **OTHER** |  |
| Boolean | BIT SMALLINT NUMERIC(1,0) DECIMAL(1,0) CHAR(1) VARCHAR(1) FLOAT |
| Picture, Binary, List, Row, Object, Item reference | VARBINARY(blobsize) BINARY(blobsize) LONGVARBINARY(blobsize)  BLOB(blobsize) Where blobsize is SessObj.$blobsize |

#### JDBC to Omnis

|  |  |
| --- | --- |
| JDBC Data Type | Omnis Data Type |
| **CHARACTER** |  |
| CHAR(n) VARCHAR(n) LONGVARCHAR(n) CLOB(n) | Character(n) |
| **DATE/TIME** |  |
| DATE | Short date 1980 |
| TIME | Short time |
| TIMESTAMP | Date time (#FDT) |
| **NUMBER** |  |
| SMALLINT | Integer 32 bit |
| INTEGER | Integer 32 bit |
| BIGINT | Integer 64 bit |
| DECIMAL(p,s) NUMERIC(p,s)  REAL FLOAT DOUBLE | Number floating dp |
| **OTHER** |  |
| BIT | Boolean |
| BINARY VARBINARY LONGVARBINARY BLOB | Binary |

## Amazon SimpleDB DAM

The Amazon DAM (DAMAZON) allows you to access the SimpleDB from Amazon Web Services LLC. According to Amazon, “SimpleDB is a highly available, scalable, and flexible non-relational data store that offloads the work of database administration. Developers simply store and query data items via web services requests, and Amazon SimpleDB does the rest.” For further information about Amazon SimpleDB, please refer to the Amazon SimpleDB website:

1. **General information**  
   <http://aws.amazon.com/simpledb>
2. **Developers Guide**  
   <http://docs.amazonwebservices.com/AmazonSimpleDB/latest/DeveloperGuide/>

This section also discusses various topics which differentiate cloud-based connectivity from traditional RDBMSs and the impact this has on the various properties and methods.

### Dependencies

The Amazon DAM has runtime dependencies on several other dynamic libraries which must be present on your system’s library search path before the DAM can be used. When a DAMAZON session object is created, the DAM attempts to locate and resolve the symbols it needs from each of the external libraries.

If one or more symbol references cannot be resolved, these are reported to the Omnis trace log as warnings, $logon() is disabled and you should not attempt to call session or statement methods, otherwise a crash may occur.

The additional files required by the Amazon DAM for each platform are as follows:

**Windows**

libcurl.dll (requires msvcr90.dll)

libeay32.dll (requires msvcrt.dll)

libxml2.dll (requires iconv.dll & zlib1.dll)

**macOS**

libcurl.dylib (where libcurl.dylib -> /usr/lib/libcurl.4.dylib, for example)

libcrypto.dylib (where libcrypto.dylib -> /usr/lib/libcrypto.0.9.7.dylib, for example)

libxml2.dylib (where libxml2.dylib -> /usr/lib/libxml2.2.dylib, for example)

**Linux**

libcurl.so (/usr/lib/libcurl.so)

libcrypto.so (/usr/lib/libcrypto.so)

libxml2.so (/usr/lib/libxml2.so)

If these libraries are not present on your system, the appropriate package(s) may need to be installed or alternatively, downloaded and compiled from source. The principal libraries shown are all available under open source licence agreements.

For developers interested in downloading and compiling client libraries from source, information about each of the projects can be obtained from:

libcurl: <http://curl.haxx.se/>

libxml2: <http://xmlsoft.org/>

libcrypto/libeay32 : <http://www.openssl.org/> (Links accurate at time of publishing)

Binary releases of these libraries may also be available to download from these and other sources.

### Logging on to SimpleDB

To connect to SimpleDB, the *endpoint* required is supplied via the $logon() hostname parameter. In the case of Amazon SimpleDB, the endpoint is “**sdb.amazonaws.com**” or “**sdb.eu-west-1.amazonaws.com**” in Europe.

Your *access key id* and *secret* are supplied via the username and password parameters, for example:

###### Do SessObj.$logon(‘sdb.amazonaws.com’,’ AGIBJ5LOYFITD3BR7’,’ H/z6t3ARzuJL26uIE07 GTS1AkK+p5’) Returns #F

For other databases, the endpoint may be specified using http syntax, for example:

###### Do SessObj.$logon( ‘http://www.remoteserver.com/?’,’user\_1’,’password’) Returns #F

If the hostname parameter is omitted, i.e. substituted with a comma, the DAM uses sdb.amazonaws.com by default.

### Meta Data

SimpleDB does not provide information about tables, columns and indexes in the same way as traditional relational databases. Instead, *domains* can be likened to tables, *items* can be likened to rows and *attributes* can be likened to columns. This has an impact on the behavior of the following meta-data methods:

|  |  |
| --- | --- |
|  |  |
| $tables() | StatObj.$tables() returns a list of available domain names in the TableOrView column of the result set. Other result columns can be ignored as SimpleDB does not support views. |
| $columns() | StatObj.$columns(cDomain) returns meta data information about the specified domain. This information is specific to SimpleDB and is returned via the DamInfoRow column of the result set. Other result columns can be ignored. |
| $indexes() | StatObj.$indexes() is not implemented since SimpleDB handles indexing automatically. |

The information returned by $columns() for a domain is summarised as follows:

|  |  |
| --- | --- |
|  |  |
| Timestamp | The date and time when metadata was calculated in Epoch (UNIX) time. |
| ItemCount | The number of all items in the domain. |
| AttributeNameCount | The number of unique attribute names in the domain. |
| AttributeValueCount | The number of all attribute name/value pairs in the domain. |
| ItemNamesSizeBytes | The total size of all item names in the domain, in bytes. |
| AttributesValuesSizeBytes | The total size of all attribute values, in bytes. |
| AttributeNamesSizeBytes | The total size of all unique attribute names, in bytes. |

### SimpleDB Attributes and Multi-Values

Unlike Relational databases, SimpleDB attributes support multiple values. For example:

|  |  |  |  |
| --- | --- | --- | --- |
| *Domain* | *Item* | *Attribute* | *Value* |
| Suits | Gents Formal Suit | Colour | Navy |
| Suits | Gents Formal Suit | Colour | Black |
| Suits | Gents Formal Suit | Colour | Grey |

In addition, SimpleDB effectively supports only a single data type: Character. All data inserted into and retrieved from SimpleDB will be character data optionally encoded as UTF-8 bytes. Once fetched into Omnis, data can be assigned to typed variables as required. Such data will be automatically converted to the appropriate data type where possible.

Each item fetched from SimpleDB can potentially have a different number of attributes and attribute names. This prevents the use of Omnis Schema classes with SimpleDB since these require rigid column names and types. When dragging a schema class onto a SimpleDB session in the Omnis SQL Browser, all that can sensibly be achieved is to create a domain with the supplied *table name.*

SimpleDB does not support SQL in the traditional sense. You cannot use $prepare() & $execute() or $execdirect() to execute INSERT, UPDATE or DELETE statements as these are not supported. Instead, these statement methods can be used only to execute SELECT statements conforming to the SimpleDB SELECT syntax.

### Creating a Domain

To manually create a domain (analogous to a *table*), use the StatObj.$createdomain() method. For example:

###### Do StatObj.$createdomain(‘Project810’) Returns #F

### Inserting Data

To insert items and attributes into SimpleDB, use the StatObj.$putattrib() method.

Each call to $putattrib() inserts a new attribute-value pair into the specified domain item. (There is no need to create the item before inserting an attribute, the item is created implicitly). Since SimpleDB supports multiple attribute values, you can assign several different values to the same attribute if required. Duplicate values are ignored. For example:

###### Do StatObj.$putattrib(‘Project810’,’Materials’,’Tools’,’13mm Wrench’) Returns #F

###### Do StatObj.$putattrib(‘Project810’,’Materials’,’Tools’,’Quick release clamps’) Returns #F

If many attributes are to be inserted, it may be preferable to assign the domain name to the StatObj.$domain property and the item name to the StatObj.$item property. These parameters can subsequently be omitted in calls to $putattrib()- and any of the other statement methods discussed below. The above example becomes:

###### Do StatObj.$domain.$assign(‘Project810’) Do StatObj.$item.$assign(‘Materials’) Do StatObj.$putattrib(, ,’Tools’,’13mm Wrench’) Returns #F Do StatObj.$putattrib(, ,’Tools’,’Quick release clamps’) Returns #F

### Deleting Data

To delete items, attributes and values from SimpleDB, use the StatObj.$delete() method.

#### Deleting Values

To delete a specific attribute value, the domain, item, attribute name and value should be specified. For Example:

###### Do StatObj.$delete(‘Project810’,’Materials’,’Timber’,’50x50x2.4m pse’) Returns #F

#### Deleting Attributes

To delete an attribute including all of its values, the domain, item and attribute name only should be specified. For example:

###### Do StatObj.$delete(‘Project810’,’Materials’,’Timber’) Returns #F

#### Deleting Items

To delete an entire item including all its attributes and values, the domain and item name only should be specified. For example:

###### Do StatObj.$delete(‘Project810’,’Materials’) Returns #F

#### Deleting a Domain

StatObj.$delete() cannot be used to delete a domain. To do this- use StatObj.$deletedomain(). This method should be used with caution as it will permanently delete all items, attributes and values contained in the domain before removing the domain itself. For example:

###### Do StatObj.$deletedomain(‘Project810’) Returns #F

### Replacing Data

Whereas $putattrib() is used to append new attributes and values, StatObj.$replaceattrib() is used to replace all values for a specified attribute with the supplied single value. For example:

###### Do StatObj.$replaceattrib(‘Suits’,’Gents Formal Suit’,’Colour’,’Navy only’) Returns #F

### Fetching Data

The Amazon DAM uses Amazon SELECT statements to fetch multiple items. These are issued using the statement object’s $prepare(), $execute() and $execdirect() methods in a similar way to traditional SQL SELECT statements. The general form of a SimpleDB SELECT statement is as follows:

###### select output\_list from domain\_name [where expression] [sort\_instructions][limit limit]

The output\_list can be:

1. \* (all attributes)
2. itemName() (the item names only)
3. count(\*)
4. An explicit list of attributes (attribute1,..., attributeN)

For further information on the SELECT statement syntax, please refer to Amazon SimpleDB Developer Guide.

Items in the result set are returned one row-at-a-time. StatObj.$resultspending indicates whether there is a further item each time a call to StatObj.$fetch() is made and StatObj.$itemcount is initially set to the number of items in the response. The destination list or row variable is automatically redefined each time $fetch() is called. For example:

###### Do StatObj.$execdirect(‘select \* from Suits where stocklevel > 1’) Returns #F Repeat   Do StatObj.$fetch(lvRow)   … Until StatObj.$resultspending = kFalse

#### Retrieving an Item

You can retrieve all attributes for a specific item using the StatObj.$getall() method. The result set (a single row) generated by this call is returned using $fetch(). For example:

###### Do StatObj.$getall(‘Suits’,’Gents Suits’) Returns #F Do StatObj.$fetch(lvRow)

#### Retrieving Item Names

You can retrieve the names of items contained within a domain by calling the StatObj.$getitems() method. The result is returned as a single item containing a single attribute. The item names will be returned either as a comma-separated list or as a single column list- as dictated by the $attribcsv property.   
A SELECT where-clause may be optionally specified if required, in which case only the names of items which satisfy the expression will be returned. For example:

###### Do StatObj.$getitems( ,”where Colour like ‘Red%’”) Returns #F Do StatObj.$fetch(lvItems)

#### Retrieving an Attribute

You can retrieve the contents of a specific attribute using the StatObj.$getattrib() method. The result set (a single row containing a single column) generated by this call is also returned using $fetch(). For example:

###### Do StatObj.$getattrib(‘Project810’,’Materials’,’Tools’) Returns #F Do StatObj.$fetch(lvRow)

#### Handling Multiple Values

When fetching data, each row returned to Omnis represents one item from the specified domain. Item attributes containing multiple values are handled in one of two ways; either as single-column lists or as comma-separated values as dictated by the StatObj.$attribcsv property.

When $attribcsv is set to kTrue (the default), rows fetched from SimpleDB will be defined with Character columns. Attributes (columns) with multiple values will be returned as a string of comma-separated values.

When $attribcsv is set to kFalse, rows fetched from SimpleDB will contain single-column lists in each column. Each single-column list will contain one row for each attribute value.

#### Handling Multiple Attributes

You can put, delete and replace several attribute values at once using the StatObj.$putmany(), StatObj.$deletemany() and StatObj.$replacemany() methods. The attribute-value pairs to be processed are supplied via a list variable defined with two character columns. Column 1 contains the attribute names, column 2 contains the corresponding values. For example:

###### Do myList.$define(lvChar1, lvChar2) Do myList.$add(‘Tools’,’Posidrive screwdriver’) Do myList.$add(‘Tools’,’Metal hammer’) Do myList.$add(‘Charges’,’1½ hours labour’) Do StatObj.$putmany( , , myList) Returns #F

You can retrieve the values of multiple attributes using the StatObj.$getmany() method. The attribute names to be retrieved are supplied via a *single*-column list, for example:

###### Do myList.$define(lvChar1) Do myList.$add(‘Tools’) Do myList.$add(‘Materials) Do myList.$add(‘Charges’) Do StatObj.$getmany( , , myList) Returns #F

Each subsequent call to $fetch() returns a row containing separate attribute- either as a comma-separated-value or as a single column list, as dictated by $attribcsv.

#### Handling Multiple Items

When executing queries, the StatObj.$itemcount property is set to the number of items in the response- implying that each call to $fetch() retrieves one item.

When the response contains only attribute values, $itemcount will be set to zero.

#### Handling Multiple Requests

The SimpleDB DAM uses the transaction management features of the DAM interface to allow multiple requests to be executed as a combined batch of requests. To enable multiple-execution, the SessObj.$transactionmode property should set to kSessionTranManual.

In this mode, actions such as $createdomain(), $putattrib(), $getattrib(), $replacemany() and $execdirect() are accepted unconditionally into a queue. Nothing is sent to or received from the database until a SessObj.$commit() is executed, at which point each request is submitted in turn.

Unlike single request execution, every multiple request generates a response. Although actions to put, create, replace and delete attributes will return empty responses, this enables any errors and execution information associated with each action to be returned. For example:

###### Do cSess.$begin() Do cStat.$putattrib(,,'Materials','White Paint') Returns #F Do cStat.$putattrib(,,'Materials',’Cement 25Kg’) Returns #F Do cStat.$replacemany(,,lvAttribList) Returns #F Do cStat.$execdirect(‘select \* from Project810') Returns #F Do cSess.$commit() Returns #F

#### Handling Multiple Responses

When in manual transaction mode, each call to $commit() generates one or more responses. The number of responses available is returned via the SessObj.$responses property.

When $commit() is executed, StatObj.$itemcount and StatObj.$columncount are set to reflect the number of items and attributes in the initial response.

Items/attributes from the response are then retrieved using one or more calls to $fetch(). When all items/attributes from the current response have been retrieved, the StatObj.$endofresponse property is set to kTrue at which point, $itemcount and $columncount are also set to reflect the next response.

When fetching an empty response, note that $endofresponse will effectively remain set to kTrue. If the corresponding action generated an error, then StatObj.$nativeerrorcode and StatObj.$nativeerrortext will be set accordingly. Otherwise, the empty response (and empty row) can be discarded.

When all responses have been retrieved, the $resultspending property is set to kFalse, otherwise $resultspending remains set the kTrue while there are still responses waiting.

It is safe to abandon and/or replace multiple requests before executing them by simply calling SessObj.$begin() or changing the transaction mode back to kSessionTranAutomatic. You can also discard pending responses in this way.

$rollback() is not supported by the SimpleDB DAM- this has no effect.

### Machine Utilization

Amazon SimpleDB measures usage of remote resources (and hence the charge it imposes on the end-user) in terms of “box usage”. Each action sent to the database incurs a box usage- quoted as a decimal fraction of one hour. StatObj.$boxusage returns the box usage for each action which generates a response.

The *session* object also has a $boxusage property which accumulates a total box usage for the open connection. When retrieving multiple responses, the box usage for each response is received (and added) in turn.

$boxusage may not be supported by all Simple databases in which case, the value remains set to zero.

### Read Consistency

Amazon SimpleDB supports two types of read consistency, defined as follows:

1. **Eventually Consistent Reads**   
   the eventual consistency option maximizes your read performance (in terms of low latency and high throughput). However, an eventually consistent read (using Select or GetAttributes) might not reflect the results of a recently completed write (using PutAttributes, BatchPutAttributes, DeleteAttributes). Consistency across all copies of data is usually reached within a second; repeating a read after a short time should return the updated data.
2. **Consistent Reads**   
   in addition to eventual consistency, Amazon SimpleDB also gives you the flexibility and control to request a consistent read if your application, or an element of your application, requires it. A consistent read (using Select or GetAttributes with ConsistentRead=true) returns a result that reflects all writes that received a successful response prior to the read.

The Amazon DAM implements this functionality using the $consistentread session property. When set to kFalse (the default setting), the eventual consistency option is used. When set to kTrue, all $getattrib() and SELECT statement results are fetched using consistent reads.

### Conditional Puts and Deletes

The PutAttributes and DeleteAttributes API calls used by Amazon SimpleDB support conditional put and delete operations which enable you to insert, replace or delete values for one or more attributes of an item if the existing value of an attribute matches the value you specify. If the value does not match or is not present, the update is rejected. Conditional Puts/Deletes are useful for preventing lost updates when different sources write concurrently to the same item.

The Amazon DAM implements this functionality using the $whereclause statement property. This property affects all *put, replace* and *delete* attribute calls and accepts a SQL-style where clause of the form:

###### “where <name> [= <value>] [exists|does not exist]”

<name> and <value> can be literal values; in which case they must be double-quoted, or bind variables. Double quotes inside literal values should be escaped using \”. For example:

###### Do cStat.$whereclause.$assign(‘where “Color” = “Light Brown”’) Do cStat.$whereclause.$assign(‘where “Undo” does not exist’) Do cStat.$whereclause.$assign(‘where “Project \”X\”” = @[lvChar]’)

Once bound, variable values should be assigned before each call to $putattrib(), $delete(), etc:

###### Do cStat.$whereclause.$assign(‘where “Name” = @[lvChar]’) Calculate lvChar as “Brookes” Do cStat.$putattrib(‘StockDB’,’Supplier1’,’Frequency’,’Daily’) Returns #F Calculate lvChar as “Robinson” Do cStat.$delete(‘StockDB’,’Supplier2’,’Frequency’) Returns #F

Currently, the *exists* condition may only be specified if both <name> and <value> attributes are also specified. To use *does not exist*, only the <name> attribute should be specified.

Subsequent calls to put, replace or delete attributes return kFalse if the condition is not met.

$whereclause is not affected by $clear(). To remove the where condition for a statement object; assign $whereclause to an empty string.

### Session Properties

|  |  |
| --- | --- |
| Property | Meaning |
| $boxusage | Returns the cumulative total of remote machine resources consumed since the session connected. Collects box-usages from statement methods as well as box-usages from multiple actions (manual transactions). Read-only. |
| $consistentread | If set to kTrue, all read operations (e.g. $getattrib() & $fetch()) are executed guaranteeing that the results of recent updates are seen immediately. If set to kFalse, the default (faster) eventual consistency option is used. |
| $responses | Returns the number of responses generated by the last call to $commit(). Applies to manual transaction mode only. Read-only. |
| $transactionmode | Used to implement multiple request processing. When set to kSessionTranAutomatic each request is sent to the database immediately. When set to kSessionTranManual, requests and queued until a $commit() is called. |

### Session Methods

|  |  |
| --- | --- |
| Method | Description |
| $begin() | Initialises/clears multiple responses in preparation for execution of a new batch of requests. Manual transaction mode only. |
| $commit() | Executes a batch of statements and retrieves multiple responses from the database. Manual transaction mode only. |

### Statement Properties

|  |  |
| --- | --- |
| Property | Meaning |
| $attribcsv | If set to kTrue (the default), attributes with multiple values are returned as comma-separated values, i.e. the fetched row will be defined with character columns. If set to kFalse, attributes will be returned as single column lists, i.e. the fetched row will contain a single column list in each column. |
| $boxusage | For statement methods which generate a response from the database, $boxusage returns the portion of a machine hour used to complete a particular request. See SessObj.$boxusage. Read-only. |
| $domain | The current domain name. $domain will be used with various statement methods if set. Statement methods which require a domain parameter will assume this value if the method parameter is omitted. |
| $endofresponse | Returns kTrue if the last item/attribute of the current response has been fetched in which case, $boxcount, $itemcount and $columncount are set to reflect the next response. Read-only. |
| $item | The current item name. As with $domain, $item will be used with various statement methods if set. Statement methods which require an item name will assume this value if the method parameter is omitted.  When retrieving an item list from the database, $item is also set to the name of the last item to be fetched. |
| $itemcount | Returns the number of items in the current response. Returns zero if the response contains only attribute information. Read-only. |
| $resultspending | Returns kTrue while there are still items/attributes waiting to be fetched from one or more responses. Read-only. |
| $whereclause | Affects all put, replace and delete attribute methods. This property accepts a SQL-style where clause of the form: “where <name> [= <value>] [exists|does not exist]” <name> and <value> can be literal values; in which case they must be double-quoted, or bind variables. Double quotes inside literal values should be escaped using \” |

### Statement Methods

|  |  |
| --- | --- |
| Method | Description |
| $createdomain() | StatObj.$createdomain([cDomainName]) creates a domain with the specified name. $createdomain() uses the value of StatObj.$domain if the parameter is omitted in which case, $domain must be predefined. Returns kTrue on success, kFalse otherwise. |
| $delete() | StatObj.$delete([cDomain],[cItem],[cAttrib],[cValue]) deletes an item, attribute or value from the specified domain. If cDomain or cItem are omitted, the values of StatObj.$domain and StatObj.$item are assumed in which case, $domain and $item must be predefined.  If cAttrib and cValue are omitted, the entire item is deleted.  If cValue is omitted, the specified attribute is deleted.  Otherwise, the specified value only is deleted from the attribute.  Returns kTrue on success, kFalse otherwise. |
| $deletedomain() | StatObj.$deletedomain([cDomain]) deletes the specified domain and all associated items/attributes. Warning: no further confirmation is sought before the domain is permanently deleted. Returns kTrue on success, kFalse otherwise. |
| $deletemany() | StatObj.$deletemany([cDomain],[cItem],lAttribs) deletes one or more values from the domain item. The attribute-value pairs are supplied via lAttribs, which should be defined with two character columns. Column 1 contains the attribute name, Column 2 contains the corresponding value to be removed. Returns kTrue on success, kFalse otherwise. |
| $getall() | StatObj.$getall([cDomain],[cItem]) executes a query to return all attributes belonging to the specified item. The result of the query is retrieved by calling StatObj.$fetch(). Returns kTrue on success, kFalse otherwise. |
| $getattrib() | StatObj.$getattrib([cDomain],[cItem],cAttrib) executes a query to retrieve the value(s) associated with the specified attribute. The result of the query is retrieved by calling StatObj.$fetch(). Returns kTrue on success, kFalse otherwise. |
| $getitems() | StatObj.$getitems([cDomain],[cWhere]) executes a query to retrieve the item names contained within the specified domain. If cWhere is specified, the text is appended to the SELECT statement. The result of the query is obtained by calling StatObj.$fetch(). Returns kTrue on success, kFalse otherwise. $getitems() is not supported by all database vendors. |
| $getmany() | StatObj.$getmany([cDomain],[cItem],lAttribs) executes a query to retrieve one or more named attributes from the domain item. The attribute names are supplied via lAttribs, which should be defined with a single character column. The result of the query is retrieved by calling StatObj.$fetch(). Returns kTrue on success, kFalse otherwise. |
| $putattrib() | StatObj.$putattrib([cDomain],[cItem],cAttrib,cValue) inserts a new attribute. If cAttrib already exists, the new value is appended to the existing value(s), otherwise a new attribute-value pair is created. Returns kTrue on success, kFalse otherwise. |
| $putmany() | StatObj.$putmany([cDomain],[cItem],lAttribs) inserts one or more values into the domain item. The attribute-value pairs are supplied via lAttribs, which should be defined with two character columns. Column 1 contains the attribute name, Column 2 contains the corresponding value. Returns kTrue on success, kFalse otherwise. |
| $replaceattrib() | StatObj.$replaceattrib([cDomain],[cItem],cAttrib,cValue) replaces all values for the specified attribute with the specified value. Existing values are deleted. Returns kTrue on success, kFalse otherwise. |
| $replacemany() | StatObj.$repalcemany([cDomain],[cItem],lAttribs) replaces one or more attributes in the domain item. The attribute-value pairs are supplied via lAttribs, which should be defined with two character columns. Column 1 contains the attribute name, Column 2 contains the new value. Existing values are deleted. Returns kTrue on success, kFalse otherwise. |

### Implementation Notes

#### Bind Variables

Queries issued using $execute() and $execdirect() may contain bind variables- for example in the where clause of the SELECT statement. The DAM inlines variable values into the SQL text each time $execute() is called, placing single quotes around each value. Values containing single quotes are escaped by adding a second single quote for each occurrence. For example:

###### Calculate lVar as “Katharine O’Hara” Do StatObj.$execdirect(“select \* from Customers where Name = @[lVar]”) Returns #F

becomes

###### select \* from Customers where Name = ‘Katharine O’’Hara’

#### Multiple Statement Objects

The SimpleDB API does not facilitate statement isolation- only session isolation. This means that each session object may spawn only one statement object.

An attempt to spawn a second statement object will result in an error.

#### Remote Procedures

SimpleDB does not support remote procedures, views or triggers. These are features of traditional relational databases.

#### Binary Data

SimpleDB attributes support only character data of maximum length 1024 bytes and are not suitable for storing binary data directly. A better approach (the intended approach) is that attribute values be used to store URLs or unique identifiers for pictures, files and other media which exist externally to the database.