Jet Database Engine
ODBC Connectivity White Paper

Neil Black
Jet Program Management
Stephen Hecht
Jet Development

July 19, 1993

TABLE OF CONTENTS

Overview

This document covers the portion of the Microsoft Jet Database Engine which deals with ODBC data. It will discuss how Jet uses ODBC and how, in turn, the Microsoft Access user interface uses Jet. Much of this document also applies to any application that uses Jet (in particular Visual Basic 3.0). The discussion pertains only to Jet Version 1.1 (and Access 1.1), and does not indicate where Jet 1.1 improves over Jet 1.0. Nor does this document address intentions for future versions of Jet and Access. Before reading this document, you should have a general understanding of ODBC and the ODBC API. For further details on ODBC and the ODBC API, please consult the ODBC Programmer's Reference.

Jet is designed around several basic concepts, which include:

Transparent access to data -- One of the main features of Jet is that it provides transparent access to any data in your environment regardless of its location and format. The data may be on PC-based DBMS such as Paradox or dBase, or it may be in enterprise systems like SQL Server or Oracle. Regardless of where the data is, Jet makes it all look the same. You don't need to know the format of the data or where it resides.

Keyset-driven model -- Jet is built around a keyset-driven cursor model. This means that data is retrieved and updated based on key values. The keyset model introduces complexities in how Jet operates against ODBC data sources. (The traditional relational database environment uses a dataset-driven model; that is, the data in a result set is thought of as one set of records with no way of directly addressing any particular record.)

ODBC API Usage

All of the ODBC API functions used by Jet are defined by ODBC to be either at either the Core or Level 1 level of API conformance. In order for an ODBC driver to be usable with Jet, the following ODBC APIs must be supported:

SQLAllocConnect
SQLAllocEnv
SQLAllocStmt
SQLCancel

SQLColumns
SQLDescribeCol
SQLDisconnect
SQLDriverConnect
SQLError
SQLExecDirect
SQLExecute
SQLFetch
SQLFreeConnect
SQLFreeEnv
SQLFreeStmt
SQLGetData
SQLGetInfo
SQLGetTypeInfo
SQLNumResultCols
SQLParamData
SQLPrepare
SQLPutData
SQLRowCount
SQLSetConnectOption
SQLSetParam
SQLSetStmtOption
SQLSpecialColumns
SQLStatistics
SQLTables
SQLTransact

Configuration Options for ODBC Connectivity

CONFIGURATION TABLE

The server table "MSysConf", is a Jet-specific server-based configuration table, with the following structure:

Column Name	Datatype	Description
Config	SMALLINT	The number of the configuration option
chValue	VARCHAR(255)	The text value of the configuration option

nValue	INTEGER	The integer value of the configuration option
Comment	VARCHAR(255)	Description of the configuration option

This table's existence is purely optional. Just after connecting to a server, JET executes a query to read its contents; if any errors occur, Jet ignores them, and assumes that MSysConf does not exist.

Currently, only one option is defined: Config = 101. If the corresponding nValue is non-zero, it is ignored. But if nValue = 0, Jet will never store userid and password information in tables attached from this server. The attach check-box "Save login ID and password locally" will be ignored. Users will be forced to type a userid and password upon first using the attached table.

This option was created so that Database Administrators concerned about security can set this option to eliminate the possibility of unauthorized users gaining access to data through using another person's computer.

The query Jet uses to read this table contains

SELECT ... FROM MSysConf ...

so it must be publicly accessible using exactly this syntax, if it exists at all. For example, on a server that supports multiple databases, MSysConf may or may not exist in any given database.

CONFIGURATION INITIALIZATION FILE

The configuration file for Access is msaccess.ini; for Visual Basic, it is vb.ini; for a Visual Basic application, it is determined by the application. The following entries affect Jet's use of ODBC and server data. The "RmtTrace" entry belongs in the [Debug] section of the .ini file; all others reside in the [ODBC] section.

ENTRY	VALUE	EFFECT
RmtTrace	0	Use asynchronous query execution if possible (default)
	8	Trace ODBC API calls into file "odbcapi.txt"

	16	Force synchronous query execution
	24	Trace ODBC API calls; force asynchronous query execution
TraceSQLMode	1	Trace SQL Jet sends to ODBC into file "sqlout.txt"
	0	No Jet-level SQL tracing (default)
QueryTimeout	S	Cancel queries that don't finish in S seconds (default: 60)
LoginTimeout	S	Cancel login attempts that don't finish in S seconds (default: 20)
ConnectionTimeout	S	Close cached connections after S seconds idle time (default: 600)
AsyncRetryInterval	M	Ask server "Is query done?" every M milliseconds (default: 500)
AttachCaseSensitive	0	Attach to first table matching specified name (default)
	1	Attach only to table exactly matching specified name

SnapshotOnly	0	Call SQLStatistics at attach time, to allow dynasets (default)
	1	Don't call SQLStatistics, forces snapshots
AttachableObjects	string	List of server object types to allow attaching to (default: 'TABLE','VIEW','SYSTEM TABLE', 'ALIAS','SYNONYM')

Connection Management

Microsoft Access offers several advanced data access features, such as:

- simultaneous browsing of multiple tables and queries, including "background" query execution
- directly updating tables and queries during browsing
- forms controls (list-boxes, subforms, etc.) may be based on tables and queries.

Depending on the capabilities of a server and the corresponding ODBC driver, Access may require multiple connections to implement such features. Two server/driver attributes are most important in this respect.

ACTIVE STATEMENTS

An "active" statement is a query whose results have not been completely fetched from the server. Some servers/drivers do not allow any other statements to be executed on a single connection if there is an "active" statement on that connection. In this case, Jet may use multiple connections (for example, when updating a record before the entire dynaset is fetched). The alternatives--discarding unfetched results, or forcing completion of the active statement before allowing updates--would be too disruptive to users. Other servers allow multiple partially-fetched statements on a single connection. In this case, Jet uses a single connection for all server interaction. Jet asks the ODBC driver for the SQL_ACTIVE_STATEMENTS info value to determine whether multiple active statements are supported.

CURSOR COMMIT/ROLLBACK BEHAVIOR

Jet maintains several internal cursors in support of dynaset operations (e.g. update/delete/data-fetch). For efficiency, these cursors are kept in a "prepared" state. A "prepared" state means that there is a query available in a persistent state which can be re-executed without having to re-state the query. Also, as noted above, some cursors may be "active", i.e., half-fetched. Servers/drivers differ in how transactions affect all prepared/active cursors on a connection. Because Jet wraps data modifications in transactions, Jet takes steps to insulate itself from these effects.

Jet identifies the "cursor behavior" to use by analyzing the most limiting behavior of two ODBC info values: SQL_ACTIVE_STATEMENTS and SQL_CURSOR_COMMIT_BEHAVIOR.

- cursor behavior = 2: cursors are not affected by transactions. Jet takes no special precautionary actions.

- cursor behavior = 1: prepared cursors remain prepared, but unfetched results on active cursors are discarded. Jet insulates active cursors by pretending that SQL_ACTIVE_STATEMENTS = 1. In this way, transactions are never done on connections with active statements.

- cursor behavior = 0: all statements (prepared or active) are destroyed completely. This is so disruptive that Jet simply prevents transactions altogether, by considering the database to be read-only.

As a special case, if the driver indicates that transactions are not supported at all (the SQL_TXN_CAPABLE info value), then Jet ignores the "cursor behavior" value, and skips the above analysis. The database is then considered to be read-only.

Connecting to SQL Server: An Example

To better illustrate the use of SQL_ACTIVE_STATEMENTS and SQL_CURSOR_COMMIT_BEHAVIOR, here is an example of how this functionality works when using the Microsoft SQL Server ODBC Driver.
Jet calls the ODBC to get the appropriate information from the driver. The SQL Server driver returns SQL_ACTIVE_STATEMENTS = 1 and SQL_CURSOR_COMMIT_BEHAVIOR = 1. Jet will, therefore, consider the "cursor behavior" of the driver to be 1. Because SQL_ACTIVE_STATEMENTS = 1 means that there can only be one active statement per connection, Jet will require two connections when performing dynaset operations (see "Dynasets v. Snapshots "below) against SQL Server data.

CONNECTION SHARING

Jet multiplexes ODBC connections internally as much as possible. After accounting for transaction effects and active statement limits (as described above), Jet shares connections based on connect strings; two connect strings are considered equal only if both:

1. The DSN value in both connect strings match, and
2. Either the DATABASE value in both connect strings match, or neither connect string has a DATABASE value.

CONNECTIONS CACHING/AGING

Jet maintains connections even when they are not explicitly in use, to avoid constantly disconnecting and reconnecting. This is invisible to the user. The number of idle connections maintained depends on the value of SQL_ACTIVE_STATEMENTS:

- If 1, then 2 idle connections are maintained.
- If greater than 1, then 1 idle connection is maintained.

During idle time, these cached connections are aged, and eventually closed down. This is also invisible to the user, and when the connection is needed again, reconnection is automatic. The ConnectionTimeout value in the [ODBC] section of the .ini file controls how long Jet keeps these connections. It defaults to 600 seconds. Regardless, when the application exits, all connections are closed down.

AUTHENTICATION

When you use an attached table without stored userid and password, or if the stored userid and password are no longer valid, Jet will attempt to login using the userid and password used to login the local Jet database; this can be quite convenient if local and remote userids and passwords are kept consistent. If this login attempt fails, you will be prompted for a userid and password by the ODBC driver's login dialog box, which will not let you change any other dialog fields. Once you logon to a remote server, Jet remembers the userid and password entered until the application exits, so you aren't prompted for it every time re-connection is necessary. This cached userid and password only apply to the remote database you originally logged onto with it; if you connect to another server/database, you'll be prompted for the userid and password that apply there.

Due to connection-sharing, once you establish a connection to a server using a given userid and password, you'll retain that identity even if you use attached tables with different userids and passwords stored in them. If you need varying levels of security on multiple tables, you should configure the server's security so that each individual user has the access rights desired, rather than design your application around multiple identities.

Attaching Tables

In Access, links to tables in an ODBC data source can be created; these links are called *attached tables*. Attaching ODBC tables allows you to use them transparently within Access, but to implement this transparency, Jet must ask the ODBC driver for a great deal of information about the table, and cache it locally. This process can be expensive and complex:

After establishing a connection to the desired data source, Jet calls the ODBC API function SQLTables to obtain a list of tables (and other similar objects) in the ODBC data source. These are presented in a list (excluding system tables, unless you set "Show System Objects" to "Yes" in the View Options dialog). When you select one, Jet calls SQLColumns, SQLStatistics, SQLSpecialColumns, and various ODBC info functions to acquire information about the selected table.

UNIQUE INDEXES ("BOOKMARKS")

To allow updating of attached ODBC tables, Jet creates *dynasets* over them. To do so, there must be a unique index on the table (if not, Jet creates a *snapshot*, which is not updatable). The unique key values of a row are also called the row's *"bookmark"*, since they uniquely identify the row, and allow direct access to the row.
During attachment, Jet elects the first unique index (if any) returned by SQLStatistics to be the "primary" index--it's key columns will comprise the bookmark. SQLStatistics returns Clustered, Hashed, and other indexes, in that order, and alphabetically within each group. Thus, Jet can be forced to elect a particular unique index as "primary" by renaming the index such that it appears first alphabetically.

Jet does not call SQLSpecialColumns(SQL_BEST_ROWID) , and makes no attempt to use a server's "native record identifier" (e.g. Oracle's "rowid") in lieu of a unique index. This, because the longevity of such identifiers varies among servers, and after inserting a new record, there is no efficient, unambiguous way for Jet to receive the new record identifier.

A server view may be attached, but will be treated exactly like an attached table with no indexes. Thus, an attached view, and any query based on one, will be a non-updatable snapshot. Server-based Stored Procedures may not be attached, since these do not resemble tables and views closely enough.

FLOATING POINT DATA IN THE BOOKMARK

A small amount of precision loss can sometimes occur when interacting with some servers, during type conversion of floating point data, i.e., data with digits to the right of the decimal place. The actual difference is slight enough to be inconsequential, but if the data forms part of a table's bookmark, then Jet may think the row has been deleted ("#Deleted" appears in an Access datasheet/form). This is because Jet asked the server for the row by its key values, but no exact match was found (due to precision loss). Jet cannot distinguish this situation from that of a genuine record deletion by another user.
If this occurs, and there is another unique index on the table that does not involve floating-point data, you should re-attach the table, forcing Jet to elect the other unique index as "primary" (as described above).

In most cases, there is not a one-to-one correspondence between the datatypes supported by Jet and the datatypes supported by a given server. But to allow transparent access, Jet must choose an "effective" type for each column in an attached table. How the ODBC driver maps server-specific types to the ODBC-defined standard types is dependent on the implementation of the driver. The following only describes the mappings between ODBC standard types and Jet datatypes.

How ODBC Datatypes are mapped to Jet types

When attaching a table, Jet calls SQLColumns to enumerate ODBC column information for each column in the table. For each column in the table, SQLColumns returns:

fSqlType	ODBC data type
lPrecision	ODBC precision of column
wScale	ODBC scale of column

For documentation on ODBC types, and ODBC's concept of "precision" and "scale", see Appendix D of the *ODBC Programmer's Reference*.

Jet maps these three values to a Jet datatype. This is the datatype stored in the attached table definition, and it is what the user sees. The ODBC type information is saved, per column, and fed back into ODBC whenever Jet "uses" the column (SELECTing, UPDATEing, INSERTing the column, and parameterizing queries by it).

The type mapping is done as follows:

Jet Datatype	Access Datatype
SQL_BIT	Yes/No
SQL_TINYINT SQL_SMALLINT	Number -- Size: Integer
SQL_INTEGER	Number -- Size: Long Integer

SQL_REAL	Number -- Size: Single
SQL_FLOAT SQL_DOUBLE	Number -- Size: Double
SQL_TIMESTAMP SQL_DATE SQL_TIME	DateTime
SQL_CHAR SQL_VARCHAR	if lPrecision <= 255, then Text (Field Size = lPrecision) if lPrecision > 255, then Memo
SQL_BINARY SQL_VARBINARY	if lPrecision <= 255, then Binary (Field Size = lPrecision) if lPrecision > 255, then OLE Object
SQL_LONGVARBINARY	OLE Object
SQL_LONGVARCHAR	Memo

SQL_DECIMAL
SQL_NUMERIC
	if wScale = 0, then
	if lPrecision <= 4, then Number -- Size: Integer
	if lPrecision <= 9, then Number -- Size: Long Integer
	if lPrecision <= 15, then Number -- Size: Double
	if wScale > 0, then
	if lPrecision <= 15, then Number -- Size: Double

Special cases for SQL Server/Sybase:
if lPrecision = 19 and wScale = 4, then Currency
if lPrecision = 10 and wScale = 4, then Currency

Anything not covered above is mapped to Text(Field Size = 255)

How Jet Datatypes are mapped to ODBC Datatypes

When executing a "SELECT INTO" query with an ODBC destination (this includes File Export in Access), Jet maps each source column type to a destination column type. A "CREATE TABLE" statement and multiple "INSERT" statements are sent to the server using these destination types. Jet calls SQLGetTypeInfo to get ODBC type info for all datatypes supported by the backend. A collection of internal data structures is built, describing the type info in a Jet-digestible format. The type mapping is described in the table below. In the mapping below, replace "SQL_SMALLINT" with "SQL_NUMERIC(5,0)" if SQL_SMALLINT is not supported by the server. Replace "SQL_INTEGER" with SQL_NUMERIC(10,0) if SQL_INTEGER is not supported. Replace "SQL_VARCHAR" with "SQL_CHAR" if SQL_VARCHAR is not supported by the server. If SQL_CHAR is also not supported, the query fails.

Jet Datatype	**ODBC Datatype**
Yes/No	SQL_BIT, if supported, else SQL_SMALLINT, if supported, else SQL_INTEGER, if supported, else SQL_VARCHAR(5)
Number --Size: Byte Number --Size:Integer	SQL_SMALLINT, if supported, else SQL_INTEGER, if supported, else SQL_VARCHAR(10)
Number --Size: Long Integer	SQL_INTEGER, if supported, else SQL_VARCHAR(20)
Currency	SQL_DECIMAL(19,4), if SQL Server/Sybase, else SQL_FLOAT, if supported, else

	SQL_VARCHAR(30)
Number --Size:Single	SQL_REAL, if supported, else SQL_FLOAT, if supported, else SQL_VARCHAR(30)
Number --Size:Double	SQL_FLOAT, if supported, else SQL_VARCHAR(40)
DateTime	SQL_TIMESTAMP, if supported, else SQL_VARCHAR(40)
Text(Field Size)	SQL_VARCHAR(MIN(Field Size,ServerMax))
Binary(Field Size)	SQL_VARBINARY(MIN(Field Size,ServerMax)), if supported, else query fails
Memo	SQL_LONGVARCHAR(ServerMax), if supported, else SQL_VARCHAR(2000), if ServerMax >= 2000, else query fails
OLE Object	SQL_LONGVARBINARY(ServerMax), if supported, else SQL_VARBINARY(2000), if ServerMax >= 2000, else query fails

DATA RETRIEVAL

As explained in the section "Datatype Mapping", at attach time, Jet chooses a Jet-datatype for each column in the attached table. When fetching data for this column, Jet must sometimes convert the data into the assigned Jet-datatype. If this conversion fails, the value is treated as NULL. This should happen rarely, because Jet intentionally chooses a Jet-datatype conservatively, choosing Text when no other Jet type has a large enough value range.

Jet and Access have no internal or user interface provisions for handling zero-length text values and NULL text values differently. Therefore, a zero-length text value fetched from a server is treated as if a NULL value had been fetched.

EXPORT (MAKE TABLE QUERIES)

The File + Export command in Access uses a Make Table query to export to an ODBC data source. A Make Table query sends a CREATE TABLE statement to the server, followed by a series of INSERT statements, one per row exported. No indexes are created on the new server table, so if it is immediately attached, it will support only read-only snapshots. You must manually create a unique index on the new table before attaching it, if you wish to update the data.
When constructing the CREATE TABLE statement, Jet replaces all non-SQL-standard characters on table and column names with underscores. So exporting a table named "Sales Jan-Mar" will produce a table named "Sales_Jan_Mar" on the server. However, no check is made for exceeding the server's maximum name length. You may need to shorten very long table and column names before exporting.

If the driver supports an identifier quoting character, Jet surrounds the table and column names in the CREATE TABLE statement with this character. Other applications that do not do automatic identifier quoting may have difficulties accessing the new table, especially if the server is case-sensitive regarding identifier names. For example, if you use a simple, command-line oriented SQL interface to double-check your exported data, you may need to explicitly quote the new table's name and column names.

Dynasets vs. Snapshots

When Jet executes a query, the result set returned is either a *dynaset* or a *snapshot*. A dynaset is a live, updatable view of the data in the underlying tables. Changes to the data in the underlying tables are reflected in the dynaset, and changes to the dynaset data are immediately reflected in the underlying tables. A snapshot is a non-updatable, unchanging view of the data in the underlying tables. The result sets for dynasets and snapshots are populated in different manners.

RESULT SET POPULATION

A snapshot is populated by executing a query that pulls back all the selected columns of the rows meeting the query's criteria. A dynaset, on the other hand, is populated by a query that selects only the bookmark (primary key) columns of each qualifying row. These queries are called "population queries". In both cases, these result sets are stored in memory (overflowing to disk if very large), allowing you to scroll around arbitrarily.
Access is optimized to return answers to you as quickly as possible; as soon as the first screenful of result data is available, Access paints it. The remainder is fetched as follows:

- User scrolling: many user actions (e.g. page down, go to last record, search) require Access to partially or completely populate the query's result set. A snapshot fetches all data up to the position scrolled to; a dynaset fetches bookmarks (primary keys) up to that point, then fetches a small amount of data surrounding that position (see below for details).
- Idle time: while you are inactive, Access populates the query's result set in the background. This allows faster operations when you become active again. A snapshot fetches and stores all selected columns, a dynaset fetches and stores only bookmarks, and no other data.

When the "population" query reaches the end of the result set, a snapshot does no further data fetching; a dynaset does no more key fetching, but will continue to fetch clusters of rows based on those bookmarks, as you scroll around (see below). In addition, if a connection is needed solely for this key-fetching query, it is closed, unless either:

1. It is parameterized. The connection is maintained to allow fast re-query (for subforms and parameterized combo-boxes).
2. This would counteract connection-caching, as described above.

DATA FETCHING

When rows of data are needed (e.g. to paint a datasheet), a snapshot has the data available locally. A dynaset, on the other hand, has only keys, and must use a separate query to ask the server for the data corresponding to those bookmarks. Jet asks the server for clusters of rows specified by their bookmarks, rather than one at a time, to reduce the querying traffic.

The dynaset behind an Access datasheet/form does in fact cache a small window of data (roughly 100 rows surrounding the current record). This reduces slightly the "liveness" of the data, but greatly speeds moving around within a small area. The data can be refreshed quickly with a single keystroke, and is periodically refreshed by Access, during idle time. This contrasts with a snapshot, which caches the entire result data set, and cannot be refreshed except by completely re-executing the query.

In addition to background key-fetching, a dynaset also fills its 100-row data window during idle time. This allows you to page up or down "instantly" once or twice, provided you give Access a least a little time to breathe.

PERFORMANCE IMPLICATIONS

Snapshots and dynasets differ in several performance characteristics, due to their different methods of retrieving and caching data. Several points are worth noting:

- Snapshots are faster to open and scroll through than dynasets. If your result set is small, and you don't need to update data or see changes made by other users, use a snapshot. Set the form's "Allow Updating" property to "No Tables" to force the form to run on a snapshot. In Basic, use the CreateSnapshot method.
- For larger result sets, a dynaset is faster and more efficient. For example, moving to the end of a snapshot requires the entire result set to be downloaded to the client. But a dynaset only downloads the bookmark columns, then fetches the last screenful of data corresponding to those keys.
- Dynaset open time and scrolling speed are affected most negatively by:

1. How many columns you select. Select only the columns you need; outputting all columns using Table.* is more convenient, but slower.
2. How many of the query's tables are output. Sometimes joins are used simply as restrictions, and don't need to be output at all.

- When a dynaset fetches the data for a given set of keys, Memo columns are not fetched unless visible on the screen. If scrolling causes them to become visible, they are fetched at that point. You can improve performance by designing your form so that, by default, Memo columns are not visible. Either place the Memo off the right/bottom edge of the screen, or add a button that renders the Memo visible when pushed. In any case, Memos are cached within the dynaset caching window, once fetched.
- OLE Objects are never fetched in bunches, nor are they stored in the dynaset caching window, since they tend to be quite large. When a row is displayed, the OLE Objects are fetched, if visible. However, the current row's OLE Objects are cached, so that simple screen-repainting does not

require re-fetching.

ASYNCHRONOUS QUERY EXECUTION

Jet executes ODBC queries asynchronously, if this is supported by the ODBC driver, the network software, and the server. This allows you to cancel a long-running query in Access, or to switch to another task in Windows while the query runs on the server. Jet asks the server if the query is finished every M milliseconds, where M is configurable, and defaults to 500 milliseconds. When you cancel a query (or simply close a query before all results have been fetched), Jet calls the ODBC function SQLCancel. When SQLCancel is called, any pending results are discarded and control is returned to the user. However, some servers (or their network communication software) do not implement an efficient query-canceling mechanism, so you may still have to wait some time before regaining control.

Asynchronous processing may cause unpredictable results with some network libraries and some servers. These network libraries are often more robust when operating synchronously, owing chiefly to the added complexities of handling multiple asynchronous connections. Client applications are often written to operate fully synchronously, even if interactive; this is simpler to implement and test. If you experience flakiness, you can force Jet to operate synchronously by setting an .ini-file option (described in the section "Configuration"). Also notify your network/server vendor: there may be an upgrade or patch for these problems.

Jet will automatically cancel a long-running query after a configurable amount of time (the default is 60 seconds). If this happens, it does not necessarily mean that the server did not respond during that time, or that you have become disconnected. It simply means the query did not return results in the time allotted. If you know a certain query will take a very long time to execute, increase the QueryTimeout setting in the .ini-file.

OPTIMIZATION OF "FIND"

Against server data, the Edit + Find command in Access and the Find method in Basic are implemented using one of two strategies: an "optimized" find and an "unoptimized" find. The optimized version is used only if:

1. table/query is a dynaset, not a snapshot

2. the column is indexed

3. Edit+Find: Match Whole Field or Start of Field, not Any Part of Field

4. Edit+Find: Current Field, not All Fields

5. Edit+Find: not Search as Formatted

6. Basic: find restriction is column = value or column LIKE value

7. Basic: the LIKE string is "foo" or "foo*"

8. Basic: server supports the LIKE operator on text columns

   The optimized algorithm first executes a query of the form

   SELECT <bookmark-columns>
   FROM table
   WHERE <find-restriction>

   The resultant bookmarks are sought in the dynaset (which stores bookmarks, not data). Currency is positioned on the first matching bookmark, if any. To find (or not find) a matching bookmark, the dynaset may need to fetch more bookmark column values from the server.

The unoptimized algorithm simply iterates through the rows of the snapshot or dynaset, evaluating the find-restriction on each row, until a match is found, or until the end of the records is reached. Again, this may require substantial fetching from the server.

Modifying Server Data

Access users change, add, and delete server data in several ways, including:

- Direct editing in a datasheet or form
- Running "action" queries (bulk UPDATEs, etc.)
- Using Access Basic/Visual Basic data access objects

In all cases, Jet can only change/delete data in attached server tables with a unique key (a bookmark). When a row is updated/deleted in a datasheet, Jet sends an UPDATE/DELETE to the server, qualified by a WHERE clause specifying the key values for that row. This controls exactly which row is updated/deleted, and protects against inadvertent multi-row updates/deletes.

Inserting new records generally also requires the existence of a bookmark. The dynaset supporting a datasheet must keep track of newly added records (they become indistinguishable from previously-existing records). Additionally, if the query does not output all the columns comprising the bookmark, inserting new records is not allowed. There are exceptions, however; Append and MakeTable action queries do not require a unique key on the remote table.

VOLATILE PRIMARY KEYS

If another user changes a bookmark column of a row, Jet loses its handle to the record, and considers it to be deleted. (Re-executing the query will remedy this situation, provided the record still meets the query's criteria.)
Because of this, if a trigger on the server changes the key values at the time of an update/insert, Jet may fail to update/insert the row. Or Jet may successfully update/insert the row, but Access will immediately display it as "#Deleted".

COLUMNS UPDATED/INSERTED

When an update/insert is performed on a datasheet, Jet supplies values for every updatable field in the datasheet, whether or not it was changed or set explicitly. This allows Jet to use a single UPDATE/INSERT statement for all updates/inserts, rather than constructing a new statement every time. This can cause three unexpected behaviors:

- A trigger that fires when a column is changed will activate, even if the column is being "changed" to its current value. You can alter the trigger to do nothing if the old and new values match.
- Columns that do not allow NULL must be included in your query, or inserts will fail. Also, you must supply values for them, or the NULL that Jet supplies will cause an error.
- Server defaults are overridden on insert by the explicit NULL supplied by Jet.

You can force an updatable query output column to be non-updatable (and exclude it from such update/insert statements) by wrapping it in an expression, such as IntCol + 0 or StringCol & ".

If a table has a "timestamp" column, Jet prevents you from updating it manually, since the server maintains its value.

SECURITY

Jet neither enforces nor overrides server-based security. Additional client-side security may be setup on attached tables and their queries, but beyond the initial connection-time login, Jet remains strictly ignorant of server security. Security violations caused by Jet queries done in support of dynaset operations will bring up dialog boxes with server-specific error messages.

LOCKING AND CONCURRENCY

Jet does no explicit server-based locking of any kind; the server's/driver's default concurrency mechanisms are used at all times. Several points are worth noting:

- The "Record Locks" form property may only be "No Locks"; "Edited Record" is ignored, and "All Records" is illegal. Similarly, the "Exclusive" option to Basic's CreateDynaset is illegal.
- Datasheet updates are done using "optimistic" concurrency: the row is not locked during editing, but is checked for conflict at update time, by further restricting the bookmark-qualified UPDATE statement. If a "timestamp" column exists in the table--as reported by SQLSpecialColumns(SQL_ROWVER)--it is qualified with its current value. If not, all columns are qualified with their former values, excluding Memo and OLE Object columns. The former method is preferable, especially considering the precision-loss problems described above in "Floating Point Data in the Bookmark".
- Jet wraps most data-modifying operations in short transactions, but longer transactions can sometimes occur:

1. Large action queries: Jet wraps the entire bulk operation within a single transaction.
2. Multi-row datasheet operations, such as multi-row pastes and deletes are wrapped in a transaction until you confirm them.
3. Transactions in Access Basic/Visual Basic: here, you are responsible for the length and breadth of your transactions, which can be arbitrarily long.

Such long-running transactions over large amounts of data can lock out or block other users, depending on the server's concurrency model.

- Automatic idle-time population does not apply to snapshot and dynaset objects in Basic code. If you stop moving through the result set and "sit on a row" for a long time, the server may hold a lock on that row or page, depending on the server's concurrency model. Due to Jet's buffering schemes, this is no longer a concern once you reach the end of the result set. This comment also applies to list-boxes and combo-boxes based on

large server-based result sets--they also do not enjoy background population.

- When performing an ORDER BY, some servers lock all of the data involved until sorting is finished, and results are returned. This is beyond Jet's control.

Some of these caveats are relevant in any client-server environment, regardless of the front-end application. In order to be a "good citizen" in such an environment, it behooves you to make judicious use of transactions and cursors on reasonably-sized result sets, and be familiar with your server's default locking behavior.

TRANSACTIONS

Multiple concurrent transactions against dynasets against a single server are actually a single transaction, since a single connection is being used to service updates for both dynasets. You should structure your transactions so that they do not overlap; after all, transactions are intended to be atomic units.
If the server supports transactions at all, as Jet determines by calling SQLGetInfo(SQL_TXN_CAPABLE), Jet assumes only single-level support, i.e., no nesting of transactions. Therefore, if your Basic code nests transactions, only the outer-most Begin, Commit, and Rollback are actually sent to the server.

BeginTrans does not "carry into" opening a dynaset on server data; before opening the dynaset, a connection to the server may not even exist. In the following code:

```
BeginTrans
Set ds = d.CreateDynaset(...)
<data modifications using ds>
ds.Close
Rollback
```

the Rollback statement does **not** undo the changes made using the dynaset. The proper way to structure this operation is as follows:

```
Set ds = d.CreateDynaset(...)
BeginTrans
<data modifications using ds>
Rollback
ds.Close
```

Because the dynaset exists when the BeginTrans and Rollback statements are reached, Jet knows what server to pass them along to.

If you use the following sequence on remote data:

```
Set ds = d.CreateDynaset(...)
BeginTrans
<data modifications using ds>
ds.Close
```

then a Rollback is sent to the server. You must explicitly Commit these data changes before closing the dynaset. This is consistent with Jet-defined behavior on native Jet tables.

PERFORMING BULK OPERATIONS

Due to the keyset-driven model used by Jet, it is important to note how bulk operations (INSERT, UPDATE, and DELETE) are performed. First, the keyset which corresponds to the records which will be affected is built. Then the appropriate operation is performed one record at a time for reach record in the keyset.

Sending Queries to a Server

The Jet query processor supports advanced capabilities such as heterogeneous joins, queries based on other queries, and arbitrary expressions, including user-defined functions. But Jet must communicate with a server in standard SQL terms, and only refer to functionality and data on that server. For any given query, Jet must determine what portion(s) may be sent to each server involved, for remote processing. The overriding goal is to send as much of the query to the server as possible, but there are some operations that must be performed locally.

Generic query optimization techniques should not be ignored when using attached server tables. Given that Jet attempts to send as much of a query as possible to the server for evaluation, you should be familiar with the capabilities of the server. For example, equality and range restrictions should still be done on indexed fields.

IDENTIFYING REMOTE PROCESSING

The query compiler generates an execution plan for a query in the form of a tree of operations, where the leaves are tables, and the root is the final query result set. Jet walks this tree from the bottom up, collapsing subtrees into SQL statements to be sent to a server. The collapsing stops when an operation:

- joins data from multiple data sources
- would not be expressible in a single SQL statement.

- is not supported on the server

Each of these conditions is covered in detail below.

PROCESSING THAT MUST BE DONE LOCALLY

The key to query performance on attached server tables is ensuring that little or no data filtering is done on the client. Client-side data processing data increases network traffic and prevents you from leveraging advanced server hardware; it effectively reduces a client/server system to a file server system. You can better optimize performance by being aware of what query operations Jet must evaluate on the client.

Heterogeneous joins

Joins spanning multiple data sources must be performed locally. Jet determines if the inputs to a join are from the same data source using the same algorithm as described above in "Connection Sharing". Some servers support multiple databases on a single server machine. Since each is a distinct ODBC "data source", Jet will not ask the server to do cross-database joins, only joins within a given database.

Operations not expressible in a single SQL statement

Jet queries may be based upon other Jet queries, allow operations such as:

- A GROUP BY over a GROUP BY or DISTINCT
- A join over one or more GROUP BYs or DISTINCTs
- Complex combinations of inner and outer joins

Jet will send to the server as much of these operations as can be expressed in a single standard SQL statement, but must perform the remaining higher-level operations locally.

Operations not supported on the server

Generally, the outputs of a query (the SELECT clause) do not affect how much of the query Jet sends to the server, and how much is processed locally. Jet selects the needed columns from the server, and locally evaluates any output expressions based upon them. The other query clauses (WHERE, ORDER BY, etc.) have a more important effect: the expressions in these other clauses determine whether or not Jet must execute them locally. Among the constructs that Jet must evaluate locally:

- Undeclared parameters. Unless you declare your parameters (via the

Query Parameters dialog in Access, or the PARAMETERS clause in Jet SQL), they are "type-less", meaning "any type of data is valid". Most servers do not have such a notion.

- Unsupported Basic operators and functions. Basic intrinsically supports many numeric, date/time, statistical, financial, and string functions. Some have server equivalents, some do not. Jet must locally evaluate any function without a server-side correspondent. Jet determines what functions/operators are supported on the server by asking the ODBC driver, via SQLGetInfo. If supported by the server and driver, Jet will send these operators and intrinsic functions remote:

General	Operators	Numeric Functions	String Functions	Aggregate Functions	Date/Time	Functions
AND	OR	ABS	LCASE	MIN	SECOND	DAY
NOT	LIKE	ATN	LEFT	MAX	MINUTE	MONTH
IN	&	COS	LEN	AVG	HOUR	YEAR
=	+	EXP	INSTR	COUNT	WEEKDAY	
< >	-	INT	LTRIM	SUM	DATEPART('ddd')	
<	*	LOG	MID		DATEPART('www')	
< =	/	MOD	RIGHT		DATEPART('yyy')	
>	IDIV	RND	RTRIM		DATEPART('mmm')	

> = MOD	SGN	SPACE		DATEPART('qqq')
BETWEEN	SIN	STRING		DATEPART('hhh')
IS [NOT] NULL	SQR	TRIM		DATEPART('nnn')
	TAN	UCASE		DATEPART('sss')
				DATEPART('ww')
				DATEPART('yyyy')

- An Access report with multiple levels of grouping and totals is not aggregated on the server, since SQL doesn't support such a concept.
- User-Defined Functions (UDFs). You can define your own functions in Basic; these never have server equivalents, so they must be evaluated locally.
- Miscellaneous unsupported functionality. Jet uses SQLGetInfo and SQLGetTypeInfo to ask the ODBC driver whether the server supports, among other things:
    1. outer joins
    2. expressions in the ORDER BY clause (as opposed to columns)
    3. the LIKE operator on Text and Memo columns
- Miscellaneous unsupported and questionable expressions:
    1. Operations involving "incompatible" types, such as
    a LIKE b + c
    2. Non-standard LIKE wildcards (such as the Access-specific '[' and '#')
    3. Intrinsic functions, if arguments have incorrect types
    4. Explicit type conversion functions (such as CInt, CDbl, etc.)
    5. Non-logical operators where logical operators should be, such

as

(a > b) AND (c + d), right-hand-side is arithmetic.

6. Logical operators where non-logical operators should be, such as

a + (b AND c) * d, using a logical result in an addition.

RESTRICTION SPLITTING

When deciding whether or not a WHERE or HAVING clause can be sent to the server, Jet dissects the restriction expression into its component conjuncts (separated by ANDs), and only evaluates locally those components that cannot be sent remote. Therefore, if you use restrictions that cannot be processed by the server, you should accompany them with restrictions that can. For example, suppose you have written a Basic function called "Foobar". The following query:

```
SELECT *
FROM huge_table
WHERE Foobar(column1) = 17
```

will cause Jet to bring back the entire table and evaluate Foobar(column1) = 17 locally; however, this query:

```
SELECT *
FROM huge_table
WHERE Foobar(column1) = 17 AND
        last_name BETWEEN 'g' AND 'h'
```

will cause Jet to send

```
SELECT *
FROM huge_table
WHERE last_name BETWEEN 'g' AND 'h'
```

to the server, bringing back only those rows that match the restriction. Jet will then locally evaluate the restriction Foobar(column1) = 17 on only those rows.

EVALUATION OF OUTPUTS

As previously mentioned, SELECT clause elements are usually evaluated locally by Jet. There are two exceptions to this rule:

- Queries with DISTINCT: provided that all SELECT clause expressions can be evaluated by the server, Jet will send the DISTINCT keyword as

well. If a SELECT expression must be evaluated by Jet locally, then so must the DISTINCT operation.

- Queries with aggregation: Jet attempts to do aggregation on the server, since this reduces the number of rows returned to the client, often drastically. For example, the query:

SELECT Sum(column1) FROM huge_table
is sent entirely to the server; a single row is returned over the network. On the other hand:

SELECT StdDev(column1) FROM huge_table
causes Jet to send "SELECT column1 FROM huge_table" to the server, retrieve every row in the table, and perform the aggregate locally. This is because StdDev is not a standard SQL aggregate.

REMOVE EXECUTION OF CROSSTAB QUERIES

Jet sends some crosstab queries to the server for evaluation; this can result in far fewer rows transferred over the network. Jet sends a simpler GROUP BY form of the crosstab, and transforms the result set into a true crosstab. But this transformation does not apply to complex crosstabs; the criteria you must meet are:

1. Row/Column Headers may not contain aggregates.
2. The Value must contain only one aggregate.
3. No user-defined ORDER BY clause.

Of course, all other reasons for forcing local processing also apply.

OUTER JOINS

In determining where to performs joins, Jet separates outer joins from inner joins, due to ambiguities inherent in mixing both join types. Thus, any query Jet sends through ODBC will have a FROM-clause containing either:

- Any number of tables, all inner joined, or
- Exactly 2 tables, outer joined.

This means that some complex queries involving inner and outer joins will not be sent completely to the server; Jet may perform some of the higher level joins locally.

There are three other conditions that cause Jet to perform an outer join locally:

1. the server does not support outer joins, or the driver does not support

the ODBC canonical syntax for specifying them.
2. the join restriction is anything other than
left_table.column = right_table.column
that is, anything but an outer join on one column.
3. the form property "Allow Updating" is set to "Any Tables"

GENERATING SQL TO SEND TO A SERVER

The SQL that Jet sends an ODBC driver is generated according to the SQL Grammar defined by ODBC. For the most part, this is standard SQL, but may contain ODBC-defined canonical escape sequences. Each ODBC driver is responsible for replacing these escape sequences with back-end specific syntax before passing the SQL along to the server; Jet never uses back-end specific syntax.
For example, most servers support outer joins, but differ widely in their outer join syntax. Jet uses only the ODBC-defined outer join syntax:

```
SELECT Table1.Col1, Table2.Col1
FROM {oj Table1 LEFT OUTER JOIN Table2 ON
        Table1.Col1 = Table2.Col1}
```

and relies on the ODBC driver to translate this to the server-specific outer join syntax. In the case of SQL Server, this would be:

```
SELECT Table1.Col1, Table2.Col1
FROM Table1, Table2
WHERE Table1.Col1 *= Table2.Col1
```

WILDCARDS FOR THE "LIKE" OPERATOR

When using the LIKE operator, you should use the Jet wildcards ('?' for single character matching, '*' for multiple character matching), not the server-specific wildcards. Jet translates these wildcards into '_' and '%' before sending the expression to the server. The only exception is in query parameter values: since Jet forwards your parameter values to the server, they must use '_' and '%'.

OWNER AND TABLE PREFIXING

Jet prefixes column names with their table name when generating queries involving more than a single table. In a self-join, Jet generates a correlation name to use as a tablename prefix. Jet also prefixes with ownername, if there is an owner associated with the attached table; this ownername, if any, was returned by the ODBC driver's SQLTables function, at attach time.

IDENTIFIER QUOTING

Jet calls SQLGetInfo(SQL_IDENTIFIER_QUOTE_CHAR) to determine the identifier quoting character supported by the server/driver. If there is one, Jet wraps all owner, table, and column names in this character, even if this is not strictly always necessary (without knowing the keywords and special characters for a particular server, Jet cannot know if quoting is necessary for any given identifier).

HIDDEN SELECT-CLAUSE EXPRESSIONS

Some servers don't allow you to place a column or expression in the ORDER BY clause or GROUP BY clause of a query unless the same column or expression appears in the SELECT clause. Since Jet SQL has no such restriction, Jet covers for the server by invisibly adding such columns and expressions to the SELECT clause before sending the query to the server. These extra outputs are discarded when received.

JET-TO-ODBC SQL TRACING

By setting TraceSQLMode=1 in the [ODBC] section of the .ini-file, you can observe the SQL statements Jet is passing to the ODBC driver. The tracing output is written to a file named "sqlout.txt" in the current directory. Jet always appends to this file, never overwriting, so you should not leave tracing turned on indefinitely.
Details of SQL tracing output:

SQLExecDirect: <SQL-string>	Execute non-parameterized user query
SQLPrepare: <SQL-string>	Prepare parameterized query
SQLExecute:	Execute prepared, parameterized user query

(PARAMETERIZED QUERY)	
SQLExecute: (GOTO BOOKMARK)	Fetch single row based on bookmark
SQLExecute: (MULTI-ROW FETCH)	Fetch 10 rows based on 10 bookmarks
SQLExecute: (MEMO FETCH)	Fetch Memos for single row based on bookmark
SQLExecute: (GRAPHIC FETCH)	Fetch OLE Objects for single row based on bookmark
SQLExecute: (ROW-FIXUP SEEK)	Fetch single row based on some index key (not necessarily bookmark index)
SQLExecute: (UPDATE)	Update single row based on bookmark
SQLExecute: (DELETE)	Delete single row based on bookmark
SQLExecute: (INSERT)	Insert single row (dynaset mode)
SQLExecute: (SELECT INTO insert)	Insert single row (export mode)

You can generally ignore queries such as:

- SELECT nValue FROM MSysConf WHERE Config = 101

See section "Configuration" for details about this query.

- SELECT 1 WHERE 0 = 1

This query is a workaround for a bug in pre-version 4.2 SQL Server.

- SELECT c1, c2, c3... FROM table1 WHERE c1 = ?

This is the GOTO BOOKMARK query, or the ROW-FIXUP SEEK query.

- SELECT c1, c2, c3... FROM table1 WHERE c1 = ? OR c1 = ? OR ... OR c1 = ?

This is the MULTI-ROW FETCH query.

You can most easily read the tracing output if you remove the "sqlout.txt" file just before running a query. The first SQLPrepare or SQLExecDirect should correspond to your query (ignoring the queries listed above).

ODBC Specification Compliance Errors

Any error returned by Jet which is in the range of -7700 to -7799 is an ODBC Specification Compliance Error. The error indicates that a driver has failed to comply to the ODBC specification and represents a bug in the driver. Any such error should be reported to the vendor who supplied the driver. The table below contains an error number which will be returned by Jet along with two other pieces of information:

1. A description of the ODBC API call which was made including any parameter values.
2. A description of the condition which caused the error.

ERROR	ODBC Call	Condition which caused the error
-7701	SQLGetInfo(ODBC_API_CONFORMANCE)	*pcbInfoValue != 2
-7702	SQLGetInfo(ODBC_API_CONFORMANCE)	wValue < 1
-7703	SQLGetData(fCType=SQL_C_CHAR)	Call return "driver could not convert"

-7704	SQLGetTypeInfo(SQL_ALL_TYPES)	neither SQL_CHAR nor SQL_VARCHAR was returned; type support is insufficient
-7705	SQLGetTypeInfo ==> SQLNumResultCols	*pccol < 6
-7706	SQLGetTypeInfo ==> SQLGetData(TYPE_NAME)	*pcbValue <= 0
-7707	SQLGetTypeInfo ==> SQLGetData(DATA_TYPE)	*pcbValue != 2
-7708	SQLGetTypeInfo ==> SQLGetData(PRECISION)	*pcbValue != 0 or *pcbValue != 4
-7709		odbc.dll missing API function (possibly bad odbc.dll)
-7710	SQLSetParam(fSQLType=SQL_VARCHAR)	driver could not convert
-7711		UNUSED
-7712		primary key must be > 255 bytes
-7713		SQL_INVALID_HANDLE returned by ODBC API; i.e., driver claims henv/hdbc/hstmt is invalid
-7714	SQLGetTypeInfo ==> SQLNumResultCols	*pccol < 9

-7715	SQLTables ==> SQLGetData(TABLE_OWNER/TABLE_NAME)	length(ownername.tablename) > 255 bytes
-7716	SQLTables ==> SQLGetData(TABLE_NAME)	*pcbValue <= 0
-7717	SQLTables ==> SQLGetData(TABLE_TYPE)	*pcbValue <= 0
-7718	SQLTables ==> SQLGetData(TABLE_TYPE)	*pcbValue > 128
-7719	SQLStatistics ==> SQLGetData(COLUMN_NAME)	total length of columns for index > 255 bytes
-7720	SQLGetInfo(SQL_CURSOR_COMMIT_BEHAVIOR)	*pcbInfoValue != 2
-7721	SQLGetInfo(SQL_CURSOR_ROLLBACK_BEHAVIOR)	*pcbInfoValue != 2
-7722	SQLTables ==> SQLNumResultCols	*pccol < 4
-7723	SQLSpecialColumns ==> SQLNumResultCols	*pccol < 2
-7724	SQLSpecialColumns ==> SQLGetData(COLUMN_NAME)	*pcbValue <= 0
-7725	SQLGetTypeInfo ==> SQLGetData(SEARCHABLE)	*pcbValue != 2
-7726	SQLGetTypeInfo ==>	value out of range

	SQLGetData(SEARCHABLE)	
-7727	SQLColumns ==> SQLNumResultCols	*pccol < 11
-7728	SQLColumns ==> SQLGetData(TABLE_OWNER)	*pcbValue < 0
-7729	SQLColumns ==> SQLGetData(TABLE_NAME)	*pcbValue <= 0
-7730	SQLColumns ==> SQLGetData(COLUMN_NAME)	*pcbValue <= 0
-7731	SQLColumns ==> SQLGetData(DATA_TYPE)	*pcbValue != 2
-7732	SQLColumns ==> SQLGetData(PRECISION)	*pcbValue != 0 or 4
-7733	SQLColumns ==> SQLGetData(SCALE)	*pcbValue != 0 or 2
-7734	SQLColumns ==> SQLGetData(NULLABLE)	*pcbValue != 0 or 2
-7735	SQLColumns ==> SQLGetData(NULLABLE)	value out of range
-7736	SQLStatistics ==> SQLNumResultCols	*pccol < 12
-7737	SQLStatistics ==> SQLGetData(TABLE_OWNER)	*pcbValue < 0

-7738	SQLStatistics ==> SQLGetData(TABLE_NAME)	*pcbValue <= 0
-7739	SQLStatistics ==> SQLGetData(NON_UNIQUE)	*pcbValue != 2
-7740	SQLStatistics ==> SQLGetData(INDEX_QUALIFIER)	*pcbValue < 0
-7741	SQLStatistics ==> SQLGetData(INDEX_QUALIFIER/INDEX_NAME)	length(qualifier.indexname) > 255 bytes
-7742	SQLStatistics ==> SQLGetData(INDEX_NAME)	*pcbValue < 0
-7743	SQLStatistics ==> SQLGetData(TYPE)	*pcbValue != 2
-7744	SQLStatistics ==> SQLGetData(TYPE)	value out of range
-7745	SQLStatistics ==> SQLGetData(TYPE/NON_UNIQUE/INDEX_NAME)	TYPE == SQL_TABLE_STAT, but either NON_UNIQUE or INDEX_NAME is non-NULL
-7746	SQLStatistics ==> SQLGetData(TYPE/NON_UNIQUE/INDEX_NAME)	TYPE != SQL_TABLE_STAT, but either NON_UNIQUE or INDEX_NAME is NULL
-7747	SQLStatistics ==> SQLGetData(COLUMN_NAME)	*pcbValue <= 0
-7748	SQLStatistics ==> SQLGetData(COLLATION)	*pcbValue != 0 or 1

-7749	SQLStatistics ==> SQLGetData(COLLATION)	value not 'A' or 'D'
-7750	SQLGetInfo(SQL_TXN_CAPABLE)	*pcbInfoValue != 2
-7751	SQLGetInfo(SQL_TXN_CAPABLE)	value < 0 or > 2
-7752	SQLGetInfo(SQL_DATA_SOURCE_READ_ONLY)	*pcbInfoValue != 1
-7753	SQLGetInfo(SQL_DATA_SOURCE_READ_ONLY)	value not 'Y' or 'N'
-7754	SQLGetInfo(SQL_IDENTIFIER_QUOTE_CHAR)	*pcbInfoValue != 1
-7755	SQLGetInfo(SQL_IDENTIFIER_QUOTE_CHAR)	value '.' or alphanum
-7756	SQLGetInfo(SQL_STRING_FUNCTIONS)	*pcbInfoValue != 4
-7757	SQLGetInfo(SQL_NUMERIC_FUNCTIONS)	*pcbInfoValue != 4
-7758	SQLGetInfo(SQL_TIMEDATE_FUNCTIONS)	*pcbInfoValue != 4
-7759	SQLGetInfo(SQL_SYSTEM_FUNCTIONS)	*pcbInfoValue != 4

-7760	SQLGetInfo(SQL_OUTER_JOINS)	*pcbInfoValue != 1
-7761	SQLGetInfo(SQL_OUTER_JOINS)	value not 'Y' or 'N'
-7762	SQLGetInfo(SQL_EXPRESSIONS_IN_ORDERBY)	*pcbInfoValue != 1
-7763	SQLGetInfo(SQL_EXPRESSIONS_IN_ORDERBY)	value not 'Y' or 'N'
-7764	SQLGetInfo(SQL_CONCAT_NULL_BEHAVIOR)	*pcbInfoValue != 2
-7765	SQLGetInfo(SQL_CONCAT_NULL_BEHAVIOR)	value not 0 or 1