White Paper

Jet Database Engine version 3.0

ODBC Connectivity

Jet Database Engine

ODBC Connectivity



Contents

Overview

ODBC API Usage

Configuration Options for ODBC Connectivity

Configuration Table

Background Population Settings

Disabling of Password Storage via the System Registry

System Registry Settings

Connection Management

Active Statements

Cursor Commit/Rollback Behavior

Connecting to SQL Server: An Example

Connection Sharing

Connections Caching/Aging

Authentication

Linking Tables

Unique Indexes ("Bookmarks")

Floating-Point Data in the Bookmark

Datatype Mapping

How ODBC Datatypes Are Mapped to Jet Types

How Jet Datatypes Are Mapped to ODBC Types

Specific Server Examples

Data Retrieval

Export (Make Table Queries)

Dynasets Vs. Snapshots

Result Set Population

Data Fetching

Performance Implications

Asynchronous Query Execution

Optimization of Find

SQL Pass-Through Queries

Integration of Pass-through Queries

Processing Multiple result sets

Modifying Server Data

Volatile Primary Keys

Columns Updated/Inserted

Security

Locking and Concurrency

Transactions

Performing Bulk Operations

Sending Queries to a Server

Identifying Remote Processing

Processing That Must Be Done Locally

Heterogeneous Joins

Operations Not Expressible in a Single SQL Statement

Operations Not Supported on the Server

Restriction Splitting

Evaluation of Outputs

Remote Execution of Crosstab Queries

Outer Joins

Generating SQL to Send to a Server

Wildcards for the LIKE Operator

Owner and Table Prefixing

Identifier Quoting

JET-to-ODBC SQL Tracing

ODBC Specification Compliance Errors

# Summary of New ODBC Features for Jet 3.0

* Disabling of password/userid storage in linked tables via the system registry.
* Initialization settings stored in the system registry.
* Datatype mapping for new Microsoft SQL Server 6.0 datatypes - Decimal, Numeric, and the IDENTITY attribute.

Each of these items is discussed in detail below and is preceded by the  label.

# Overview

This document covers the portion of the Microsoft Jet Database Engine that deals with Open Database Connectivity data. It discusses how Jet uses ODBC and how, in turn, the Microsoft Access user interface uses Jet. The discussion pertains only to Jet version 3.0 (and the Microsoft Access for Windows ’95 version 7.0 database management system) and only occasionally indicates the areas in which Jet 3.0 improves over Jet 2.0. This document does not address intentions for future versions of Jet and Microsoft Access.

This document will be most helpful to readers with 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, including:

**Transparent access to data** Jet provides transparent access to any database in your environment, regardless of the data's location and format. The data may be on a PC–based DBMS, such as Paradox® or dBASE®, or in an enterprise system, such as Microsoft SQL Server or ORACLE®. Regardless of where the data is, Jet makes it all look like the native format.

**Keyset-driven model** Jet is built around a *keyset-driven* cursor model. This means data is retrieved and updated based on key values. The keyset model introduces complexities in how Jet operates against ODBC data sources. (Traditional relational database environments use 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 a particular record.)

# ODBC API Usage

All the ODBC API functions used by Jet are defined by ODBC to be 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

SQLBindParameter

SQLCancel

SQLColumns

SQLDescribeCol

SQLDisconnect

SQLDriverConnect

SQLError

SQLExecDirect

SQLExecute

SQLFetch

SQLFreeConnect

SQLFreeEnv

SQLFreeStmt

SQLGetData

SQLGetFunctions

SQLGetInfo

SQLGetStmtOption

SQLGetTypeInfo

SQLMoreResults (optional)

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) | A description of the configuration option |

This table's existence is purely optional. Immediately after connecting to a server, Jet executes a query to read its contents. If the table doesn't exist, no error occurs, and defaults are assumed. If the table exists, and any errors occur, Jet refuses the connection, and returns an error. The following options are defined (chValue and Comment are currently unused):

|  |  |  |
| --- | --- | --- |
| **Config** | **nValue** | **Meaning** |
| 101 | 0 | Don't allow storing user & password in linked tables |
| 101 | 1 | Allow storing user & password in linked tables (the default) |
| 102 | D | Access delays D seconds between each background chunk fetch (default=10) |
| 103 | N | Access fetches N rows on each background chunk fetch (default=100) |

### Background Population Settings

The background population options allow an administrator to control how fast Access fetches rows of a query during idle time. By setting the fetch delay high, network traffic is reduced, but read locks are left on pages longer. By setting delay lower, locking is reduced, and moving to the last record in a datasheet is speeded, but network traffic increases. The chunk size option provides an ever finer level of control.

The query Jet uses to read this table is  
 SELECT Config, nValue 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 might or might not exist in a given database.

Disabling password storage in linked tables causes the "Save password" check box on the Link Tables dialog to be grayed and unavailable. Jet will then never store userid and password information in tables linked from this server. Users will be forced to type a userid and password upon first using the linked table. This option was created to permit database administrators concerned about security to eliminate the possibility of unauthorized users gaining access to data through using another person's computer.

### Disabling of Password Storage via the System Registry

An alternate way of disabling password/userid storage for links is to use the System Registry rather than the MSysConf table. When running Microsoft Windows NT, to disable the storage of password/userid for linked tables set CachedLogonCount to a DWORD value of 0 under the registry key \\HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon. When running Microsoft Windows ’95, to disable the storage of password/userid for linked tables set DisablePwdCaching to a DWORD value of 1 under the registry key \\HKEY\LOCAL\_MACHINE\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Policies\\Network.

### System Registry Settings

When using Jet 2.0 and Microsoft Access 2.0, configuration settings were stored in an INI file called, MSACC20.INI. When using Jet 3.0 and Microsoft Access 7.0 (or any other 32-bit client of Jet’s), configuration settings are stored in the system registry. The following entries affect Jet's use of ODBC and server data. All reside in the under the registry key \\HKEY\_LOCAL\_MACHINE\Software\Microsoft\Jet\3.0\Engines\ODBC. Note: This registry location may vary based on the calling application, i.e., Access, VB, a custom app, etc.

|  |  |  |  |
| --- | --- | --- | --- |
| **Key** | **Type** | **Default** | **Description** |
| LoginTimeout | DWORD | 20 | Number of seconds to wait for a login attempt to succeed. |
| QueryTimeout | DWORD | 60 | Number of seconds to wait for a query to execute before failing. |
| ConnectionTimeout | DWORD | 600 | Number of seconds to wait before closing an idle connection. |
| AsyncRetryInterval | DWORD | 500 | Number of milliseconds to wait between retries on an asynchronous connection. |
| AttachCaseSensitive | DWORD | 0 | If False(0), will link to the first name matching the specified string in a match that isn’t case-sensitive. If True(1), requires a case-sensitive match of the name. |
| AttachableObjects | SZ | TABLE | The list of objects that Microsoft Jet will allow links to (TABLE, NEW, SYSTEMTABLE, ALIAS, SYNONYM). Values are of type REG\_SZ. |
| SnapshotOnly | DWORD | 0 | If False(0), you will get index information on linked tables to allow dynasets if possible. If True(1), Microsoft Jet will ignore index information and thereby force snapshots on all linked tables. |
| TraceSQLMode | DWORD | 0 | Initiates sending a trace of SQL statements sent to an ODBC data source to the file SQLOUT.TXT. Values are 0 (No) and 1 (Yes). The default is 0 (values are of type REG\_DWORD). This entry is interchangeable with SQLTracemode. |
| TraceODBCAPI | DWORD | 0 | Initiates sending a trace of ODBC API calls to the file ODBCAPI.TXT. Values are 0 (No) and 1 (Yes). The default is 0 (values are of type REG\_DWORD). |
| DisableAsync | DWORD | 1 | Forces synchronous query execution. Values are 0 (use asynchronous query execution if possible) and 1 (force synchronous query execution). The default is 0 (values are of type REG\_DWORD). |
| JetTryAuth | DWORD | 1 | Uses the Microsoft Access user name and password to log on to the server before prompting. Values are 0 (No) and 1 (Yes). The default is 0 (values are of type REG\_DWORD). |
| PreparedInsert | DWORD | 0 | Uses a prepared INSERT statement that inserts data in all columns. Values are 0 (use a custom INSERT statement that inserts only non-Null values) and 1 (use a prepared INSERT statement). The default is 0 (values are of type REG\_DWORD).Using prepared INSERT statements can cause Nulls to overwrite server defaults and can cause triggers to execute on columns that weren’t inserted explicitly. |
| PreparedUpdate | DWORD | 0 | Uses a prepared UPDATE statement that updates data in all columns. Values are 0 (use a custom UPDATE statement that sets only columns that have changed) and 1 (use a prepared UPDATE statement). The default is 0 (values are of type REG\_DWORD).Using prepared UPDATE statements can cause triggers to execute on columns that weren’t changed explicitly. |
| FastRequery | DWORD | 0 | Uses a prepared SELECT statement for parameterized queries. Values are 0 (No) and 1 (Yes). The default is 0 (values are of type REG\_DWORD). |

# Connection Management

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

Simultaneous browsing of multiple tables and queries, including "background" query execution.

Direct updating of tables and queries during browsing.

Forms controls (list boxes, subforms, and so on) that may be based on tables and queries.

Depending on the capabilities of a server and the corresponding ODBC driver, Microsoft Jet might 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.

If an ODBC driver supports doesn't allow multiple statements on a connection, and supports only a single connection, ever, then Jet ignores unique indexes when linking tables, in order to force snapshot-only mode, which requires only a single-connection.

### Cursor Commit/Rollback Behavior

Jet maintains several internal cursors in support of dynaset operations. For efficiency, these cursors are kept in a *prepared state.* Also, as noted above, some cursors may be active, that is, half-fetched. Servers/drivers differ in the way 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: Treated just like Cursor behavior = 1, except that after each transaction, Jet re-prepares its internal statements.

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.

#### 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 makes ODBC API calls to get the appropriate information from the driver. The SQL Server driver returns SQL\_ACTIVE\_STATEMENTS = 1, SQL\_CURSOR\_COMMIT\_BEHAVIOR = 1, and SQL\_CURSOR\_ROLLBACK\_BEHAVIOR = 1; Jet will therefore consider the cursor behavior of the driver to be 1. SQL\_ACTIVE\_STATEMENTS = 1 means only one statement can be active per connection, so Jet will require two connections when performing dynaset operations against SQL Server data. (See "Dynasets Vs. Snapshots," later in this paper.)

Dynasets with less than 100 rows to be processed with a single connection. The 100 keys are quickly fetched before the second connection is needed to support dynaset operations.

### 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 of the following criteria are met:

1. The DSN value in both connect strings match.

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

For example, if you open two large SQL Server tables, a total of three connections are required: the first to fetch keys from the first table, the second to fetch keys from the second table, and the third to do all updating and chunk-data-fetching from both tables. As soon as all keys from either table are fetched, the corresponding connection will be released. If the tables resided on an Oracle server, however, only a single connection is required to do all three tasks.

### 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 two idle connections are maintained.

If greater than 1, then one idle connection is maintained.

During idle time, these cached connections are aged and eventually closed down, even if datasheets, forms, and recordset variables are still using the connections. This is also invisible to the user, and when the connection is needed again, reconnection is silent and automatic. The ConnectionTimeout value in the system registry controls how long Jet keeps these connections open. It defaults to 600 seconds. Regardless, when the application exits, all connections are closed down. Two conditions prevent a connection from being timed out:

An un-committed transaction on the connection.

A query with unfetched results on the connection.

### Authentication

When you use a linked table without a stored userid and password, or if the stored userid and password are no longer valid, Jet will attempt to log in using the userid and password used to log in the local Jet database (this can be disabled by an .ini-file entry); this can be 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 log on to a remote server, Jet remembers the userid and password entered until the application exits, so you aren't prompted for it every time reconnection is necessary. This cached userid and password apply only to the remote database you originally logged on to 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 linked 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.

# Linking Tables

In Microsoft Access, links to tables in an ODBC data source can be created; these links are called *linked tables*. Linked ODBC tables allows you to use them transparently within Microsoft 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 linked ODBC tables, Jet creates *dynasets* over them. 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* because they uniquely identify and allow direct access to the row.

When creating a link, Jet elects the first unique index (if any) returned by SQLStatistics to be the *primary index—*its 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 (for example, Oracle's "rowid") in lieu of a unique index. 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 linked but will be treated exactly like a linked table with no indexes. Thus a linked view, and any query based on one, will be a non-updatable snapshot. However, if you know that certain columns uniquely identify rows in the view (perhaps they comprise a unique index in the underlying table), you can create a pseudo unique index on the linked table itself, by using a DDL query, such as:

CREATE UNIQUE INDEX Index1  
 ON LinkedTable (Column1, Column2)

Do not make this a SQL Pass-through query; this does not create an index on the server's table or view. But it tells Jet how to uniquely identify rows, and allows dynaset functionality such as updating. Server-based Stored Procedures may not be linked because these do not resemble tables and views closely enough.

### Floating-Point Data in the Bookmark

Because servers vary in how precise they can be in their handling of floating-point data, sometimes precision loss can occur. *Floating-point data* is defined as data with digits to the right of the decimal point. Very large or very small floating-point values might lose some accuracy when being tranferred from some servers to Jet. The actual difference is slight enough to be inconsequential, but if the data forms part of a table's bookmark, Jet might think the row has been deleted ("#Deleted" appears in a Microsoft 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 another unique index on the table does not involve floating-point data, you should re-link the table, forcing Jet to elect the other unique index as "primary" (as described in the previous section).

# Datatype Mapping

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 a linked table. How the ODBC driver maps server-specific types to the ODBC–defined standard types depends on the implementation of the driver. The following describes only the mappings between ODBC standard types and Jet datatypes.

### How ODBC Datatypes Are Mapped to Jet Types

When linking 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 datatype

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 linked 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:

|  |  |
| --- | --- |
| **ODBC Datatype** | **Microsoft 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 | DateTime |
| SQL\_TIME | Text |
| 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:  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 Types

When executing a SELECT INTO query with an ODBC destination (this includes File Export in Microsoft 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 back end. A collection of internal data structures is built, describing the type info in a Jet-digestible format. The type mapping is described in the following table.

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.

|  |  |
| --- | --- |
| **Microsoft Access 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, 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 if the server supports it. Otherwise, SQL\_VARCHAR(n), where n is the server’s maximum size for a VARCHAR if the maximum size is greater than 2000. If neither case is supported on the database server, the query fails. |
| OLE Object | SQL\_LONGVARBINARY if the server supports it. Otherwise, SQL\_VARBINARY (n), where n is the database server’s maximum size for a VARBINARY if the maximum size is greater than 2000. If neither case is supported on the database server, the query fails. |

### Specific Server Examples

When you link a SQL Server table in Microsoft Access:

|  |  |
| --- | --- |
| **SQL Server Type** | **Maps to Access Type** |
| bit | Yes/No |
| tinyint | Number (Integer) |
| smallint | Number (Integer) |
| int | Number (Long Integer) |
| real | Number (Single) |
| float | Number (Double) |
| decimal(p,s) | 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:  if lPrecision = 19 and wScale = 4, then Currency  if lPrecision = 10 and wScale = 4, then Currency |
| numeric(p,s) | 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:  if lPrecision = 19 and wScale = 4, then Currency  if lPrecision = 10 and wScale = 4, then Currency |
| smallmoney | Currency |
| money | Currency |
| smalldatetime | Date/Time |
| datetime | Date/Time |
| char(n) | Text(n) |
| varchar(n) | Text(n) |
| text | Memo |
| binary(n) | Binary(n) |
| varbinary(n) | Binary(n) |
| image | OLE Object |

 Microsoft SQL Server version 6.0 and greater supports an attribute called IDENTITY. The IDENTITY attribute can be applied to any numeric column. A column with the identity attribute is a read-only, system-maintained column which works like a Jet Counter column. Microsoft Access calls a Counter column an AutoNumber of size Long Integer.

When a record is inserted into the table which contains an IDENTITY column, the system automatically fills in the value for the IDENTITY column. If the IDENTITY attribute is set on a column of type int, smallint, or tinyint, it will be mapped to a Jet Counter column. If the IDENTITY attribute is set on a numeric column of any other type, it is mapped to the appropriate datatype as described in the table above.

When you export a Microsoft Access table to SQL Server:

|  |  |
| --- | --- |
| **Access Type** | **Maps to SQL Server Type** |
| Yes/No | bit |
| Number (Byte) | smallint |
| Number (Integer) | smallint |
| Number (Long Integer) | int |
| Number (Single) | real |
| Number (Double) | float |
| Currency | money |
| Date/Time | datetime |
| Counter | int |
| Text(n) | varchar(n) |
| Memo | text |
| OLE Object | image |

When you link an Oracle table in Microsoft Access:

|  |  |
| --- | --- |
| **Oracle Type** | **Maps to Access Type** |
| number(1-4,0) | Number (Integer) |
| number(5-9,0) | Number (Long Integer) |
| number(10-15,0) | Number (Double) |
| number(16-38,0) | Text |
| number(1-15,n) | Number (Double) |
| number(16-38,n) | Text |
| float | Number (Double) |
| date | Date/Time |
| char(n) | Text(n) |
| raw(n) | Binary(n) |
| long | Memo |
| longraw | OLE Object |

When you export a Microsoft Access table to Oracle:

|  |  |
| --- | --- |
| **Access Type** | **Maps to Oracle Type** |
| Yes/No | number(5,0) |
| Number (Byte) | number(5,0) |
| Number (Integer) | number(5,0) |
| Number (Long Integer) | number(10,0) |
| Number (Single) | float |
| Number (Double) | float |
| Currency | float |
| Date/Time | date |
| Counter | number(10,0) |
| Text(n) | char(20) |
| Memo | long |
| OLE Object | long raw |

### Data Retrieval

As explained in "Datatype Mapping," earlier in this white paper, at link time, Jet chooses a Jet datatype for each column in the linked 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 rarely happen because Jet chooses datatypes conservatively; for example, Jet chooses Text when no other Jet type has a large enough value range. Zero-length text value fetched from a server is treated as if a NULL value had been fetched.

### Export (Make Table Queries)

The Export command in the Microsoft Access File menu 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 linked, it will support only read-only snapshots. You must manually create a unique index on the new table before linking it if you want to update the data.

When constructing the CREATE TABLE statement, Jet replaces all non-SQL-standard characters on table and column names with underscores. For example, 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 might 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 might 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 might 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.

Microsoft Access is optimized to return answers to you as quickly as possible; as soon as the first screenful of result data is available, Microsoft Access paints it. The remainder is fetched as follows:

User scrolling: Many user actions (for example, page down, go to last record, and search) require Microsoft 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 and then fetches a small amount of data surrounding that position. (See the following text for details.)

Idle time: While you are inactive, Microsoft 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. You can control how quickly this idle-time population occurs, using the MSysConf server-based options table.

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 requery (for subforms and parameterized combo boxes), or

2. This would counteract connection-caching, as described above.

### Data Fetching

When rows of data are needed (for example, 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 Microsoft Access datasheet/form does in fact cache a small window of data (roughly 100 rows surrounding the current record). This slightly reduces 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 Microsoft Access during idle time. This contrasts with a snapshot, which caches the entire result data set and cannot be refreshed except by complete re-execution of 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 Microsoft Access at least a little idle time.

Microsoft Access exposes this caching mechanism via the Data Access Objects (DAO) through two Recordset properties (CacheStart and CacheSize), and a Recordset method (FillCache). These apply only to dynasets (not snapshots or pass-through queries), and only when the dynaset contains at least some ODBC data. CacheStart and CacheSize indicate the beginning and length (in rows) of the local cache, while FillCache fills the cache with remote data, fetched in chunks, rather than a single row at a time.

### 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 a Snapshot object.

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 downloads only the bookmark columns and then fetches the last screenful of data corresponding to those keys.

Dynaset open time and scrolling speed are affected most negatively by the number of columns you select and the number of the query's tables that are output. Select only the columns you need; outputting all columns using Table.\* is more convenient but slower. 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 they are visible on the screen. If scrolling causes them to become visible, they are then fetched. 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, because they tend to be quite large. When a row is displayed, the OLE objects are fetched if they are visible. However, the current row's OLE objects are cached, so 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 Microsoft Access or to switch to another task in the Windows™ operating system 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. SQLCancel discards any pending results and returns control to the user. However, some servers (or their network communication software) do not implement an efficient query-canceling mechanism, so you might still have to wait some time before regaining control.

Asynchronous processing might 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. You can force Jet to operate synchronously by setting an .ini file option (described earlier in this paper). Also notify your network/server vendor; an upgrade or patch might be available 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 query's "ODBC Timeout" property. Each query can have its own timeout setting.

### Optimization of Find

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

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

2. The column is indexed.

3. The Find command on the Edit menu: Match Whole Field or Start of Field, not Any Part of Field.

4. The Find command on the Edit menu: Current Field, not All Fields.

5. The Find command on the Edit menu: not Search as Formatted.

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

7. Basic: the LIKE string is smith or smith\*.

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

The optimized algorithm first executes a query of the following form.

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

The resulting 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 might 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.

# SQL Pass-Through Queries

A SQL Pass-through query is simply an arbitrary string (presumed to be SQL), and an ODBC connect string used to specify a server. The SQL string must be acceptable to the specific back-end it is directed at, and may do any of the following, but is not limited to:

Select data from tables, using server-specific syntax.

Change data in server tables.

Call stored procedures, which may or may not return results.

Perform administrative actions, such as add/delete user accounts.

When executing the SPT query, Jet establishes a connection to the server via the connect string (or shares an existing connection), and simply passes the SQL string directly to the ODBC driver, without interpreting it in any way. The ODBC driver generally passes it directly to the server for interpretation and execution. The server returns zero, one, or more sets of result rows, and might also return informational and warning messages. Result rows are fetched on-demand, as needed in a datasheet/form, to lighten network traffic.

### Integration of Pass-through Queries

If the SPT query returns at least one result set of rows, and you set the query property "Returns Rows" to Yes (the default), it is treated exactly like any other snapshot by Jet and Microsoft Access; you can use it in other queries, as the basis for a form, report, or list-box, and in Basic code. Given this level of flexibility, you could write a Pass-through query that called a server-based stored procedure that performed a server-specific calculation, and returned a set of rows; you could use this SPT query as the basic for a crosstab query that joined in some local data, and in turn use this crosstab query as the basic for a form or report. The only limitation to a SPT query is that the result set is not updatable.

If you set the query property "Log Messages" to Yes, Jet will log informational and warning messages generated by the SPT query to a new Jet table named "User - ##", where User is your Jet username, and ## is a number from 0 to 99. Unless you delete these tables, the number will increase by 1 each time you run the query with logging activated.

When setting the Connect property of a SPT query, you can use the "ODBC Connect String" builder to avoid typing the actual string. You'll be prompted by the ODBC driver for the server, database, userid and password. Be warned, however, that you must be able to connect to the server in order to use this builder.

### Processing Multiple result sets

Pass-through queries can return multiple result sets, but in a datasheet, form, report, or recordset variable, only the first result set is processed. The others are ignored. The only way to retrieve multiple result sets is to create a MakeTable query on top of the SPT query as follows:

SELECT MultiSPTQuery.\*  
INTO LocalResults  
FROM MultiSPTQuery

assuming that "MultiSPTQuery" is the name of the SPT query that returns multiple result sets. Do not select individual columns from the SPT query; you must select "\*", since each result set may have different columns. Result sets are placed into new tables named after your destination table, with ascending integers appended. In the example, if 3 result sets were returned, then 3 Jet tables would be created:

LocalResults first result set  
LocalResults1 second result set  
LocalResults2 third result set

Messages from such a query are logged in their own single table, if the "Log Messages" property is set to Yes on the SPT query.

# Modifying Server Data

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

Direct editing in a datasheet or form.

Running "action" queries (for example, bulk UPDATEs).

Using Microsoft Access Basic data access objects.

In all cases, Jet can change/delete only data in linked 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 multirow updates/deletes.

Inserting new records also requires the existence of a bookmark. The dynaset supporting a datasheet must keep track of newly added records, by keeping track of the keys of the new records. Therefore, if the query does not output all the columns constituting the bookmark, inserting new records is not allowed. Exceptions to these rules occur, 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.)

If a trigger on the server changes the key values at the time of an update, Jet might successfully update the row, but Microsoft Access will immediately display it as "#Deleted."

However, if a trigger on the server changes or initializes the key values at the time of an insert (for example, a trigger that simulates a "Counter" column or a column in a Microsoft SQL Server 6.0 database with the IDENTITY attribute set), Jet notices that the key has changed, and re-selects the new row, based on all other values in the new record. Provided that this re-select returns exactly one row, Jet will acquire the key values and be able to keep track of the new record. If zero rows or more than one row is returned by this re-selection, the row appears as "#Deleted".

### Columns Updated/Inserted

When an update is performed on a datasheet, Jet supplies values for every field that the user changed during the edit. When an insert is performed, Jet supplies values for every field that is not NULL. This improves upon Jet Version 1.1 in the following ways:

A trigger that fires when a column is changed will not activate if the user doesn't actually edit the column value.

Server columns that do not allow NULL values can be included in queries and forms safely, because Jet will not force a NULL value on insert.

Server defaults will not be overridden by Jet; in fact, as soon as the new row is saved, the server-supplied defaults appear in the new row in the datasheet.

If a table has a "timestamp" column (not to be confused with a "date/time" column), Jet prevents you from updating it manually because the server maintains its value.

### Security

Jet neither enforces nor overrides server-based security. Additional client-side security may be set up on linked tables and their queries, but beyond the initial connection-time login, Jet remains strictly ignorant of server security. Security violations attempted by Jet queries done in support of dynaset operations (such as illegal updates) 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 treated exactly like "No Locks"; and "All Records" is illegal. Similarly, the "Exclusive" option to the CreateDynaset method 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, excluding Memo and OLE Object columns, are qualified with their former values. The former method is preferable, especially considering the precision-loss problems described earlier in this paper, 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. Multirow datasheet operations, such as multirow pastes and deletes. These are wrapped in a transaction until you confirm them.

3. Transactions in Microsoft Access Basic: you are responsible for the length and breadth of your transactions, which can be arbitrarily long.

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 might 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 are also not populated during idle time.

When performing an ORDER BY, some servers lock all 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, you should 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 because a single connection is being used to service updates for both dynasets. You should structure your transactions so that they do not overlap; 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, that is, no nesting of transactions. Therefore, if your Basic code nests transactions, only the outermost Begin, Commit, and Rollback are actually sent to the server.

Improvements to Jet's remote transaction management now allow seamless use of server transactions in Basic code. The Jet V1.1 requirements as to sequence of operations have been lifted. BeginTrans now "carries into" opening a dynaset on server data, even if a connection to the server didn't exist before opening the dynaset. The following code works as expected:

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

It is no longer necessary to structure your code as follows (although it still works):

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

If you use the following sequence on remote data, a Rollback is not sent to the server (as in V1.1), and the server transaction remains open until you either explicitly commit it or roll it back, or until the application terminates (at which time it will be rolled back).

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

It is now also possible to nest several bulk operations in a transaction, as in the following "credit/debit" style operation:

BeginTrans  
d.Execute("UPDATE SavingsAccount  
 SET Balance = Balance - 100")  
d.Execute("UPDATE CheckingAccount  
 SET Balance = Balance + 100")  
CommitTrans/Rollback

These transaction semantics also apply to SQL Pass-through queries that modify server data, so explicit transactions within the Pass-through queries are not necessary. SQL Pass-through queries are discussed elsewhere in this paper.

### Performing Bulk Operations

Due to the keyset-driven model used by Jet, it is important to note how bulk operations (action queries, such as INSERT, UPDATE, DELETE, and MAKETABLE) are performed. First the keyset for the records that will be affected is built. Then the appropriate operation is performed, one record at a time, for each record in the keyset. Although this is slower than performing a single qualified bulk operation on the server, it allows for partially successful bulk queries as well as bulk queries that cannot be executed by the server. When this additional functionality is not required, it is often faster to use a SQL Pass-through query (discussed elsewhere in this paper).

# 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 refer only to functionality and data on that server. For any given query, Jet must determine what portions 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 some operations must be performed locally.

Generic query optimization techniques should not be ignored when using linked 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, and closed-range restrictions (column between value1 and value2) are generally better than open-range restrictions (column > value).

### 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 matches any or all of the following conditions.

It joins data from multiple data sources.

It would not be expressible in a single SQL statement.

It is not supported on the server.

Each of these conditions is covered in detail in the following text.

### Processing That Must Be Done Locally

The key to query performance on linked 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 whether the inputs to a join are from the same data source using the same algorithm (as described earlier in this paper). Some servers support multiple databases on a single server machine. Because each is a distinct ODBC data source, Jet will not ask the server to do cross-database joins—only joins within a given database.

However, if your heterogeneous join involves a local table/query with reletively few rows, and a remote table with many more rows, and the remote table's join column(s) is indexed, Jet will perform a "remote index join". Rather than fetch the entire remote table, and perform the join locally, Jet will request only rows that match the values in the local table/query, thus greatly reducing network traffic, and dramatically improving the performance of the query.

For example, if you have a local table/query containing perhaps 10 rows of Customer information, and you join it (on the CustomerId column) to a remote table of perhaps 10,000 rows of Order information, then Jet will send 10 queries to the server, of the form

SELECT Orders.OrderId  
FROM Orders

WHERE Orders.CustomerId = ?

and supply a different Customers.CustomerID on each query. This simulates the concept of an "index seek", and only retrieves exactly the rows that match. If additional restrictions on the remote table were specified (such as only Orders over 100 dollars), then these restrictions are added to the above query:

SELECT Orders.OrderId  
FROM Orders

WHERE Orders.CustomerId = ? AND Orders.Amount > 100

The decision to do a "remote index join" is made by the local cost-based optimizer, so if the remote table is small enough, Jet will simply fetch it in its entirety, and perform the join locally, on the assumption that this is faster than submitting multiple queries.

#### Operations Not Expressible in a Single SQL Statement

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

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

Operations stacked on top of UNIONs

Subqueries that mix heterogenous data

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, and so on) 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 are the following:

TOP N and TOP N PERCENT queries

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 to the server for evaluation:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **General Operators** | **Numeric Functions** | **String Functions** | **Aggregate Functions** | **Date/Time Functions** | **Conversion Functions** |
| = | ABS | ASC | MIN | DATE | CInt |
| < > | ATN | CHR | MAX | NOW | CLng |
| < | COS | INSTR | AVG | TIME | CSng |
| < = | EXP | LCASE | COUNT | SECOND | CDbl |
| > | FIX | LTRIM | SUM | MINUTE | CCur |
| > = | INT | LEFT |  | HOUR | CStr |
| AND | LOG | LEN |  | WEEKDAY | CVDate |
| OR | RND | MID |  | DAY |  |
| NOT | SGN | RTRIM |  | MONTH |  |
| LIKE | SIN | RIGHT |  | YEAR |  |
| IS NULL | SQR | SPACE |  | DATEPART('ddd') |  |
| IS NOT NULL | TAN | STR |  | DATEPART('www') |  |
| IN |  | STRING |  | DATEPART('yyy') |  |
| & |  | TRIM |  | DATEPART('mmm') |  |
| + |  | UCASE |  | DATEPART('qqq') |  |
| - |  |  |  | DATEPART('hhh') |  |
| \* |  |  |  | DATEPART('nnn') |  |
| / |  |  |  | DATEPART('sss') |  |
| IDIV |  |  |  | DATEPART('ww') |  |
| MOD |  |  |  | DATEPART('yyyy') |  |

However, if an unsupported Basic function (such as IIF, Choose, etc.) or a Domain Function (DMax, DSum, etc.) has constant arguments, it will be evaulated once, locally, and the value will be used as an implicit remote query parameter. For example, given the query

SELECT \*  
FROM RemoteTable  
WHERE RemoteColumn > DMax("LocalColumn", "LocalTable")

Jet will evaluate the DMax once, and send this query to the server, supplying the DMax result as the parameter value:

SELECT \*  
FROM RemoteTable  
WHERE RemoteColumn > ?

A Microsoft Access report with multiple levels of grouping and totals is not aggregated on the server because 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. However, if the UDF has constant arguments, it will be evaulated once, locally, and the value will be used as an implicit remote query parameter (see example above).

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. Nonstandard LIKE wildcards (such as the '[' and '#').

Note: it is not necessary to explicitly declare query parameters. Jet infers the type of an undeclared query parameter from its surrounding expression context. For example, in the query

SELECT \* FROM Customers WHERE Name LIKE [Param1] & "\*"

the type of the parameter "Param1" is inferred to be Text, whereas in the query

SELECT \* FROM Orders WHERE Price \* Quantity > [Param2]

the type of parameter "Param2" is inferred to be a number.

### 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 remotely. Therefore, if you use restrictions that cannot be processed by the server, you should accompany them with restrictions that can be processed by the server. For example, suppose you have written a Basic function called "MyFunction". The following query will cause Jet to bring back the entire table and evaluate MyFunction(column1) = 17 locally.

SELECT \*  
FROM huge\_table  
WHERE MyFunction(column1) = 17

It is better to formulate the query as follows, if possible:

SELECT \*  
FROM huge\_table  
WHERE MyFunction(column1) = 17 AND  
 last\_name BETWEEN 'g' AND 'h'

The preceding query will cause Jet to send the following to the server, bringing back only those rows that match the restriction.

SELECT \*  
FROM huge\_table  
WHERE last\_name BETWEEN 'g' AND 'h'

Jet will then locally evaluate the restriction MyFunction(column1) = 17 on only those rows.

### Evaluation of Outputs

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

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.

### Remote 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 to send the optimal amount of a crosstab query to the server are:

1. Row/Column Headers may not contain aggregates.

2. The Value must contain only one aggregate.

3. There can be no user-defined ORDER BY clause.

All other reasons for forcing local processing also apply.

### Outer Joins

In determining where to performs joins, Jet adheres to the ODBC specification that limit mixing of inner and outer joins. Thus, any query Jet sends through ODBC will have a FROM clause containing any number of inner joins, but at most one outer join (perhaps combined with some inner joins). This means that some complex queries involving multiple outer joins will not be sent completely to the server; Jet may perform some of the higher level joins locally.

Two other conditions 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 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. Even in a query parameter, Jet translates wildcards embedded in the parameter value, each time you enter it.

### 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 an owner is associated with the linked table; this ownername, if any, was returned by the ODBC driver's SQLTables function at link time.

### Identifier Quoting

Jet calls SQLGetInfo(SQL\_IDENTIFIER\_QUOTE\_CHAR) to determine the identifier quoting character supported by the server/driver. If one exists, 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 whether quoting is necessary for any given identifier).

### JET-to-ODBC SQL Tracing

By setting TraceSQLMode=1 under the \\HKEY\_LOCAL\_MACHINE\Software\Microsoft\Jet\3.0\Engines\ODBC key in the system registry, 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: (PARAMETERIZED QUERY) | Execute prepared, parameterized user 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 Config, nValue FROM MSysConf  
See the section on configuration, earlier in this paper, for details about this query.

SELECT 1 WHERE 0 = 1  
This query is a workaround for a bug in versions of SQL Server prior to version 4.2.

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

1. A description of the ODBC API call that was made, including any relevant parameter values.

2. A description of the condition that caused the error.

|  |  |  |
| --- | --- | --- |
| **Error** | **ODBC Call** | **Condition That 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 |  | Driver returned an error, but SQLError returned no error strings. |
| -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 ==> SQLGetData(SEARCHABLE) | Value out of range |
| -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 |
| -7766 | SQLGetData(SQL\_C\_BIT) | pcbValue != 1 |
| -7767 | SQLGetData(SQL\_C\_SHORT) | pcbValue != 2 |
| -7768 | SQLGetData(SQL\_C\_TIMESTAMP) | pcbValue != sizeof(TIMESTAMP\_STRUCT) |
| -7769 | SQLGetInfo(SQL\_CONVERT\_FUNCTIONS) | pcbValue != 4 |
| -7770 | SQLGetInfo(SQL\_CONVERT\_SMALLINT) | pcbValue != 4 |
| -7771 | SQLGetInfo(SQL\_CONVERT\_INTEGER) | pcbValue != 4 |
| -7772 | SQLGetInfo(SQL\_CONVERT\_FLOAT) | pcbValue != 4 |
| -7773 | SQLGetInfo(SQL\_CONVERT\_DECIMAL) | pcbValue != 4 |
| -7774 | SQLGetInfo(SQL\_CONVERT\_TIMESTAMP) | pcbValue != 4 |
| -7775 | SQLGetInfo(SQL\_CONVERT\_VARCHAR) | pcbValue != 4 |
| -7776 | SQLGetData(SQL\_C\_TIMESTAMP) | Illegal date/time value returned. |

\* To order the *ODBC Programmer's Reference,* call Microsoft Sales and Service at 1-800-227-4679 and request part number 273050v100. International callers, request part number 273050av100.

© 1996 Microsoft Corporation. All rights reserved. Printed in the United States of America.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS DOCUMENT.

Microsoft, Microsoft Access, and the Microsoft logo are registered trademarks and Visual Basic and Windows are trademarks of Microsoft Corporation.

Paradox is a registered trademark of Ansa Software, a Borland Company. dBASE is a registered trademark of Borland International, Inc. ORACLE is a registered trademark of Oracle Corporation.

Microsoft Corporation

One Microsoft Way

Redmond, WA 98052-6399

