Designing Performance-Optimized ODBC Applications

Introduction

Developing performance-oriented ODBC applications is not easy. Microsoft's *ODBC Programmer's Reference* does not provide information about system performance. In addition, ODBC drivers and the ODBC driver manager do not return warnings when applications run inefficiently.

This white paper contains some general guidelines that have been compiled by examining the ODBC implementations of numerous shipping ODBC applications. These guidelines include:

- Use catalog functions appropriately
- Retrieve only required data
- Select functions that optimize performance
- Manage connections and updates

Following these general rules will help you solve some common ODBC performance problems, such as those listed in the following table.

Problem	Solution	See Guidelines in…
Network communication is slow.	Reduce network traffic.	"Using Catalog Functions" on page 2
Evaluation of complex SQL queries on the database server is slow and can reduce concurrency.	Simplify queries.	"Using Catalog Functions" on page 2 "Selecting ODBC Functions" on page 7
Excessive calls from the application to the driver slow performance	Optimize application-to-driver interaction.	"Retrieving Data" on page 4 "Selecting ODBC Functions" on page 7
Disk input/output is slow.	Limit disk input/output.	"Managing Connections and UpdatesManaging Connections and Updates" on page 10

Using Catalog Functions

Because catalog functions, such as those listed here, are slow compared to other ODBC functions, their frequent use can impair system performance:

SQLColumns	SQLProcedureColumns
SQLColumnPrivileges	SQLSpecialColumns
SQLForeignKeys	SQLStatistics
SQLGetTypeInfo	SQLTables
SQLProcedures	SQLTablePrivileges

SQLGetTypeInfo is listed as a performance-expensive ODBC function because many drivers must query the server to obtain accurate information about which data types are supported (for example, to find dynamic types such as user-defined types).

The guidelines in this section will help you to optimize system performance when selecting and using catalog functions.

Minimizing the Use of Catalog Functions

Compared to other ODBC functions, catalog functions are relatively slow. By caching information, applications can avoid multiple executions. Although it is almost impossible to write an ODBC application without catalog functions, you can improve system performance by minimizing their use.

To return all result column information mandated by the ODBC specification, an ODBC driver may have to perform multiple queries, joins, subqueries, or unions to return the required result set for a single call to a catalog function. These particular elements of the SQL language are performance expensive.

Applications should cache information from catalog functions so that multiple executions are unnecessary. For example, call SQLGetTypeInfo once in the application and cache the elements of the result set that your application depends on. It is unlikely that any application uses all elements of the result set generated by a catalog function, so the cached information should not be difficult to maintain.

Avoiding Search Patterns

Passing null arguments or search patterns in catalog functions generates time-consuming queries. In addition, network traffic potentially increases because of unwanted results. Always supply as many non-null arguments to catalog functions as possible. Because catalog functions are slow, applications should invoke them efficiently. Any information that the application can send the driver when calling catalog functions can result in improved performance and reliability.

For example, consider a call to SQLTables where the application requests information about the table "Customers." Often, this call is coded as shown,

using the fewest non-null arguments necessary for the function to return success:

```
rc = SQLTables (NULL, NULL, NULL, NULL, "Customers", SQL_NTS,
   NULL);
```

A driver processes this SQLTables call into SQL that looks like this:

```
SELECT ... FROM SysTables
   WHERE TableName = 'Customers' UNION ALL
SELECT ... FROM SysViews
   WHERE ViewName = 'Customers' UNION ALL
SELECT ... FROM SysSynonyms
   WHERE SynName = 'Customers' ORDER BY ...
```

In our example, the application provides scant information about the object for which information was requested. Suppose three "Customers" tables were returned in the result set: the first table owed owned by the user, the second owned by the sales department, and the third is a view created by management. It may not be obvious to the end user which table to choose. If the application had specified the OwnerName argument in the SQLTables call, only one table would be returned and performance would improve. Less network traffic would be required to return only one result row and unwanted rows would be filtered by the database. In addition, if the TableType argument was supplied, the SQL sent to the server can be optimized from a three-query union into a single Select statement as shown:

```
SELECT ... FROM SysTables WHERE TableName = 'Customers' and
   Owner = 'Beth'
```

Using a Dummy Query to Determine Table Characteristics

Avoid using SQLColumns to determine characteristics about a table. Instead, use a dummy query with SQLDescribeCol.

Consider an application that allows the user to choose the columns that will be selected. Should the application use SQLColumns to return information about the columns to the user or prepare a dummy query and call SQLDescribeCol?

Case 1: SQLColumns Method

```
rc = SQLColumns (... "UnknownTable" ...);
// This call to SQLColumns will generate a query to
// the system catalogs... possibly a join
// which must be prepared, executed, and produce
// a result set
rc = SQLBindCol (...);
rc = SQLExtendedFetch (...);
// user must retrieve N rows from the server
// N = # result columns of UnknownTable
// result column information has now been obtained
```

Case 2: SQLDescribeCol Method

```
// prepare dummy query
rc = SQLPrepare (... "SELECT * from UnknownTable
   WHERE 1 = 0" ...);
// query is never executed on the server - only prepared
rc = SQLNumResultCols (...);
for (irow = 1; irow <= NumColumns; irow++) {
   rc = SQLDescribeCol (...)
   // + optional calls to SQLColAttributes
   }
// result column information has now been obtained
// Note we also know the column ordering within the
// table!  This information cannot be
// assumed from the SQLColumns example.
```

In both cases, a query is sent to the server, but in Case 1, the query must be evaluated and form a result set that must be sent to the client. Clearly, Case 2 is the better performing model.

To complicate this discussion, let us consider a database server that does not support natively preparing a SQL statement. The performance of Case 1 does not change, but the performance of Case 2 improves slightly because the dummy query is evaluated before being prepared. Because the Where clause of the query always evaluates to FALSE, the query generates no result rows and is processed without accessing table data. Again, for this situation, Case 2 outperforms Case 1.

Retrieving Data

To retrieve data efficiently, return only the data that you need, and choose the most efficient method of doing so. The guidelines in this section will help you to optimize system performance when retrieving data with ODBC applications.

Retrieving Long Data

Unless it is necessary, applications should not request long data (SQL_LONGVARCHAR and SQL_LONGVARBINARY data) because retrieving long data across a network is slow and resource-intensive.

Most users don't want to see long data. If the user does need to see these result items, the application can query the database again, specifying only long columns in the select list. This method allows the average user to retrieve the result set without having to pay a high performance penalty for network traffic.

Although the best method is to exclude long data from the select list, some applications do not formulate the select list before sending the query to the ODBC driver (that is, some applications `select * from table_name ...`). If the select list contains long data, the driver must retrieve that data at fetch

time, even if the application does not bind the long data in the result set. When possible, use a method that does not retrieve all columns of the table.

Reducing the Size of Data Retrieved

To reduce network traffic and improve performance, you can reduce the size of any data being retrieved to a manageable limit by calling SQLSetStmtOption with the SQL_MAX_LENGTH option.

Although eliminating SQL_LONGVARCHAR and SQL_LONGVARBINARY data from the result set is ideal for performance optimization, sometimes, long data must be retrieved. When this is the case, remember that most users do not want to see 100 KB, or more, of text on the screen. What techniques, if any, are available to limit the amount of data retrieved?

Many application developers mistakenly assume that if they call SQLGetData with a container of size x that the ODBC driver only retrieves x bytes of information from the server. Because SQLGetData can be called multiple times for any one column, most drivers optimize their network use by retrieving long data in large chunks and then returning it to the user when requested. For example:

```
char CaseContainer[1000];
...
rc = SQLExecDirect (hstmt, "SELECT CaseHistory FROM Cases
   WHERE CaseNo = 71164", SQL_NTS);
...
rc = SQLFetch (hstmt);
rc = SQLGetData (hstmt, 1, CaseContainer,(SWORD)
 sizeof(CaseContainer), ...);
```

At this point, it is more likely that an ODBC driver will retrieve 64 KB of information from the server instead of 1000 bytes. In terms of network access, one 64-KB retrieval is less expensive than 64 retrievals of 1000 bytes. Unfortunately, the application may not call SQLGetData again; therefore, the first and only retrieval of CaseHistory would be slowed by the fact that 64 KB of data must be sent across the network.

Many ODBC drivers allow you to limit the amount of data retrieved across the network by supporting the SQL_MAX_LENGTH attribute. This attribute allows the driver to communicate to the database server that only *x* bytes of data are relevant to the client. The server responds by sending only the first *x* bytes of data for *all* result columns. This optimization substantially reduces network traffic and improves client performance. The previous example returned only one row, but if you consider the case where 100 rows are returned in the result set—the performance improvement would be substantial.

Using Bound Columns

Retrieving data using bound columns (SQLBindCol) instead of SQLGetData reduces the ODBC call load and improves performance.

Consider the following code fragment:

```
rc = SQLExecDirect (hstmt, "SELECT <20 columns>
   FROM Employees WHERE HireDate >= ?", SQL_NTS);
do {
rc  = SQLFetch (hstmt);
// call SQLGetData 20 times
} while ((rc == SQL_SUCCESS) || (rc == SQL_SUCCESS_WITH_INFO));
```

Suppose the query returns 90 result rows. In this case, more than 1890 ODBC calls are made (20 calls to SQLGetData x 90 result rows + 91 calls to SQLFetch).

Consider the same scenario that uses SQLBindCol instead of SQLGetData:

```
rc = SQLExecDirect (hstmt, "SELECT <20 columns>
   FROM Employees WHERE HireDate >= ?", SQL_NTS);
// call SQLBindCol 20 times
do {
rc  = SQLFetch (hstmt);
} while ((rc == SQL_SUCCESS) || (rc == SQL_SUCCESS_WITH_INFO));
```

The number of ODBC calls made is reduced from more than 1890 to about 110 (20 calls to SQLBindCol + 91 calls to SQLFetch). In addition to reducing the call load, many drivers optimize how SQLBindCol is used by binding result information directly from the database server into the user's buffer. That is, instead of the driver retrieving information into a container and then copying that information to the user's buffer, the driver simply requests the information from the server be placed directly into the user's buffer.

Using SQLExtendedFetch Instead of SQLFetch

Use SQLExtendedFetch instead of SQLFetch to retrieve data. The ODBC call load decreases (resulting in better performance), and the code is less complex (resulting in more maintainable code).

Most ODBC drivers now support SQLExtendedFetch for forward only cursors; yet, most ODBC applications use SQLFetch to retrieve data. Again consider the previous example using SQLExtendedFetch instead of SQLFetch:

```
rc = SQLSetStmtOption (hstmt, SQL_ROWSET_SIZE, 100);
// use arrays of 100 elements
rc = SQLExecDirect (hstmt, "SELECT <20 columns>
   FROM Employees WHERE HireDate >= ?", SQL_NTS);
// call SQLBindCol 1 time specifying row-wise binding
do {
rc = SQLExtendedFetch (hstmt, SQL_FETCH_NEXT, 0, &RowsFetched,
   RowStatus);
} while ((rc == SQL_SUCCESS) || (rc == SQL_SUCCESS_WITH_INFO));
```

Notice the improvement from the previous examples. The initial call load was more than 1890 ODBC calls. By choosing ODBC functions carefully, the number of ODBC calls made by the application has now been reduced to 4 (1 SQLSetStmtOption + 1 SQLExecDirect + 1 SQLBindCol + 1 SQLExtendedFetch). In addition to reducing the call load, many ODBC

drivers retrieve data from the server in arrays, further improving the performance by reducing network traffic.

For ODBC drivers that do not support SQLExtendedFetch, the application can enable forward-only cursors using the ODBC cursor library (call SQLSetConnectOption using SQL_ODBC_CURSORS or SQL_CUR_USE_IF_NEEDED). Although using the cursor library does not improve performance, it should not be detrimental to application response time when using forward-only cursors (no logging is required). Furthermore, using the cursor library when SQLExtendedFetch is not supported natively by the driver simplifies the code because the application can always depend on SQLExtendedFetch being available. The application does not require two algorithms (one using SQLExtendedFetch and one using SQLFetch).

Choosing the Right Data Type

Retrieving and sending certain data types can be expensive. When you design your schema, select the data type that can be processed most efficiently. For example, integer data is processed faster than floating-point data. Floating-point data is defined according to internal database-specific formats, usually in a compressed format. The data must be decompressed and converted into a different format so that it can be processed by the wire protocol.

Processing time is shortest for character strings, followed by integers, which usually require some conversion or byte ordering. Processing floating-point data and timestamps is at least twice as slow as processing integers.

Selecting ODBC Functions

The guidelines in this section will help you select which ODBC functions will give you the best performance.

Using SQLPrepare/SQLExecute and SQLExecDirect

Use SQLExecDirect for queries that will be executed once and SQLPrepare/SQLExecute for queries that will be executed multiple times.

ODBC drivers are optimized based on the perceived use of the functions that are being executed. SQLPrepare/SQLExecute is optimized for multiple executions of statements that use parameter markers. SQLExecDirect is optimized for a single execution of a SQL statement. Unfortunately, more than 75% of all ODBC applications use SQLPrepare/SQLExecute exclusively.

Consider the case where an ODBC driver implements SQLPrepare by creating a stored procedure on the server that contains the prepared statement. Creating stored procedures involves substantial overhead, but the statement can be executed multiple times. Although creating stored

procedures is performance-expensive, processing is minimal because the query is parsed and optimization paths are stored when the procedure is created.

Using SQLPrepare/SQLExecute for a statement that is executed only once results in unnecessary overhead. Furthermore, applications that use SQLPrepare/SQLExecute for large single-execution query batches exhibit poor performance. Similarly, applications that always use SQLExecDirect do not perform as well as those that use a logical combination of SQLPrepare/SQLExecute and SQLExecDirect sequences.

Using Arrays of Parameters

Passing arrays of parameter values, for bulk insert operations, for example, with SQLPrepare/SQLExecute and SQLExecDirect can reduce the ODBC call load and network traffic. To use arrays of parameters, the application calls SQLSetStmtAttr with the following attribute arguments:

- SQL_ATTR_PARAMSET_SIZE sets the array size of the parameter.
- SQL_ATTR_PARAMS_PROCESSED_PRT assigns a variable filled by SQLExecute, which contains the number of rows that are actually inserted.
- SQL_ATTR_PARAM_STATUS_PTR points to an array in which status information for each row of parameter values is returned.

NOTE: With ODBC 3.*x*, calls to SQLSetStmtAttr with the SQL_ATTR_PARAMSET_SIZE, SQL_ATTR_PARAMS_PROCESSED_ARRAY, and SQL_ATTR_PARAM_STATUS_PRT arguments replace the ODBC 2.*x* call to SQLParamOptions.

Before executing the statement, the application sets the value of each data element in the bound array. When the statement is executed, the driver tries to process the entire array contents using one network roundtrip. For example, let us compare the following examples, Case 1 and Case 2.

Case 1: Executing Prepared Statement Multiple Times

```
rc = SQLPrepare (hstmt, "INSERT INTO DailyLedger (...) VALUES
   (?,?,...)", SQL_NTS);
// bind parameters
...
do {
// read ledger values into bound parameter buffers
...
rc = SQLExecute (hstmt);
// insert row
} while ! (eof);
```

Case 2: Using Arrays of Parameters

```
SQLPrepare (hstmt, " INSERT INTO DailyLedger (...) VALUES
   (?,?,...)", SQL_NTS);
SQLSetStmtAttr (hstmt, SQL_ATTR_PARAMSET_SIZE, (UDWORD)100,
   SQL_IS_UINTEGER);
SQLSetStmtAttr (hstmt, SQL_ATTR_PARAMS_PROCESSED_PRT,
   &rows_processed, SQL_IS_POINTER);
// Specify an array in which to return the status of each set
// of parameters.
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAM_STATUS_PTR,
   ParamStatusArray, SQL_IS_POINTER);
// pass 100 parameters per execute
// bind parameters
...
do {
// read up to 100 ledger values into bound parameter buffers
...
rc = SQLExecute (hstmt);
// insert a group of 100 rows
} while ! (eof);
```

In Case 1, if there are 100 rows to insert, 101 network roundtrips are required to the server, one to prepare the statement with SQLPrepare and 100 additional roundtrips for each time SQLExecute is called.

In Case 2, the call load has been reduced from 100 SQLExecute calls to only 1 SQLExecute call. Furthermore, network traffic is reduced considerably.

Using the Cursor Library

If the driver provides scrollable cursors, do not use the cursor library automatically. The cursor library creates local temporary log files, which are performance-expensive to generate and provide worse performance than native scrollable cursors.

The cursor library adds support for static cursors, which simplifies the coding of applications that use scrollable cursors. However, the cursor library creates temporary log files on the user's local disk drive as it performs the task. Typically, disk input/output is a slow operation. Although the cursor library is beneficial, applications should not automatically choose to use the cursor library when an ODBC driver supports scrollable cursors natively.

Typically, ODBC drivers that support scrollable cursors achieve high performance by requesting that the DBMS server produce a scrollable result set, instead of emulating this ability by creating log files. Many applications use:

```
rc = SQLSetConnectOption (hdbc, SQL_ODBC_CURSORS,
   SQL_CUR_USE_ODBC);
```

but should use:

```
rc = SQLSetConnectOption (hdbc, SQL_ODBC_CURSORS,
   SQL_CUR_USE_IF_NEEDED);
```

Managing Connections and Updates

The guidelines in this section will help you to manage connections and updates to improve system performance for your ODBC applications.

Managing Connections

Connection management is important to application performance. Optimize your application by connecting once and using multiple statement handles, instead of performing multiple connections. Avoid connecting to a data source after establishing an initial connection.

Although gathering driver information at connect time is a good practice, it is often more efficient to gather it in one step rather than two steps. Some ODBC applications are designed to call informational gathering routines that have no record of already attached connection handles. For example, some applications establish a connection and then call a routine in a separate DLL or shared library that reattaches and gathers information about the driver. Applications that are designed as separate entities should pass the already connected HDBC pointer to the data collection routine instead of establishing a second connection.

Another bad practice is to connect and disconnect several times throughout your application to process SQL statements. Connection handles can have multiple statement handles associated with them. Statement handles can provide memory storage for information about SQL statements. Therefore, applications do not need to allocate new connection handles to process SQL statements. Instead, applications should use statement handles to manage multiple SQL statements.

On Windows, you can significantly improve performance with connection pooling, especially for applications that connect over a network or through the World Wide Web. With connection pooling, closing connections does not close the physical connection to the database. When an application requests a connection, an active connection from the connection pool is reused, avoiding the network input/output needed to create a new connection.

Connection and statement handling should be addressed before implementation. Spending time and thoughtfully handling connection management improves application performance and maintainability.

Managing Commits in Transactions

Committing data is extremely disk input/output intensive and slow. If the driver can support transactions, always turn Autocommit off.

What does a commit actually involve? The database server must flush back to disk every data page that contains updated or new data. This is not a sequential write but a searched write to replace existing data in the table. By default, Autocommit is on when connecting to a data source. Autocommit

mode usually impairs performance because of the significant amount of disk input/output needed to commit every operation.

Some database servers do not provide an Autocommit mode. For this type of server, the ODBC driver must explicitly issue a COMMIT statement and a BEGIN TRANSACTION for every operation sent to the server. In addition to the large amount of disk input/output required to support Autocommit mode, a performance penalty is paid for up to three network requests for every statement issued by an application.

Although using transactions can help application performance, do not take this tip too far. Leaving transactions active can reduce throughput by holding locks on rows for long times, preventing other users from accessing the rows. Commit transactions in intervals that allow maximum concurrency.

Choosing the Right Transaction Model

Many systems support distributed transactions; that is, transactions that span multiple connections. Distributed transactions are at least four times slower than normal transactions due to the logging and network input/output necessary to communicate between all the components involved in the distributed transaction. Unless distributed transactions are required, avoid using them. Instead, use local transactions when possible.

Using Positional Updates and Deletes

Use positional updates and deletes or SQLSetPos to update data. Although positional updates do not apply to all types of applications, developers should use positional updates and deletes when it makes sense. Positional updates (either through "update where current of cursor" or through SQLSetPos) allow the developer to signal the driver to "change the data here" by positioning the database cursor at the appropriate row to be changed. The developer is not forced to build a complex SQL statement but simply supplies the data to be changed.

In addition to making the application more maintainable, positional updates usually result in improved performance. Because the database server is already positioned on the row for the Select statement in process, performance-expensive operations to locate the row to be changed are not needed. If the row must be located, the server usually has an internal pointer to the row available (for example, ROWID).

Using SQLSpecialColumns

Use SQLSpecialColumns to determine the optimal set of columns to use in the Where clause for updating data. Often, pseudo-columns provide the fastest access to the data, and these columns can only be determined by using SQLSpecialColumns.

Some applications cannot be designed to take advantage of positional updates and deletes. These applications usually update data by forming a

Where clause consisting of some subset of the column values returned in the result set. Some applications may formulate the Where clause by using all searchable result columns or by calling SQLStatistics to find columns that are part of a unique index. These methods usually work but can result in fairly complex queries.

Consider the following example:

```
rc = SQLExecDirect (hstmt, "SELECT first_name, last_name,
     ssn, address, city, state, zip FROM emp", SQL_NTS);
// fetchdata
...
rc = SQLExecDirect (hstmt, "UPDATE EMP SET ADDRESS = ?
     WHERE first_name = ? and last_name = ? and ssn = ?
     and address = ? and city = ? and state = ?
     and zip = ?", SQL_NTS);
// fairly complex query
```

Applications should call SQLSpecialColumns/SQL_BEST_ROWID to retrieve the optimal set of columns (possibly a pseudo-column) that identifies a given record. Many databases support special columns that are not explicitly defined by the user in the table definition but are "hidden" columns of every table (for example, ROWID and TID). These pseudo-columns provide the fastest access to data because they typically point to the exact location of the record. Because pseudo-columns are not part of the explicit table definition, they are not returned from SQLColumns. To determine if pseudo-columns exist, call SQLSpecialColumns.

Consider the previous example again:

```
...
rc = SQLSpecialColumns (hstmt, ..... 'emp', ...);
...
rc = SQLExecDirect (hstmt, "SELECT first_name, last_name,
     ssn, address, city, state, zip, ROWID FROM emp", SQL_NTS);
// fetch data and probably "hide" ROWID from the user
...
rc = SQLExecDirect (hstmt, "UPDATE emp SET address = ?
     WHERE ROWID = ?", SQL_NTS);
// fastest access to the data!
```

If your data source does not contain special pseudo-columns, the result set of SQLSpecialColumns consists of the columns of the optimal unique index on the specified table (if a unique index exists). Therefore, your application does not need to call SQLStatistics to find the smallest unique index.

Conclusion

Although developing performance-optimized ODBC applications is not easy, you can improve system performance by following the general guidelines described in this white paper.

- Use catalog functions appropriately
- Retrieve only required data
- Select functions that optimize performance
- Manage connections and updates

With thoughtful design and implementation, you can ensure that your ODBC applications run more efficiently and generate less network traffic.

We welcome your feedback! Please send any comments concerning documentation, including suggestions for other topics that you would like to see, to:

docgroup@datadirect.com

FOR MORE INFORMATION

800-876-3101

Worldwide Sales

Belgium (French).....0800 12 045
Belgium (Dutch).....0800 12 046
France.....0800 911 454
Germany 0800 181 78 76
Japan 0120.20.9613
Netherlands.....0800 022 0524
United Kingdom.....0800 169 19 07
United States.....800 876 3101

DataDirect Technologies is focused on standards-based data connectivity, enabling software developers to quickly develop and deploy business applications across all major databases and platforms. DataDirect Technologies offers the most comprehensive, proven line of data connectivity components available anywhere. Developers worldwide at more than 250 leading independent software vendors and thousands of corporate IT departments rely on DataDirect® products to connect their applications to an unparalleled range of data sources using standards-based interfaces such as ODBC, JDBC™ and ADO.NET. Developers also depend on DataDirect to radically simplify complex data integration projects using XML products based on the emerging XQuery and XQJ standards. DataDirect Technologies is an operating company of Progress Software Corporation (Nasdaq: PRGS), a US$300+ million global software industry leader. Headquartered in Bedford, Mass., DataDirect Technologies can be reached on the Web at http://www.datadirect.com or by phone at +1-800-876-3101.