CHAPTER FIVE

ODBC Applications: Writing Good Code

Developing performance-optimized ODBC applications is not easy. Microsoft's *ODBC Programmer's Reference* does not provide information about performance. In addition, ODBC drivers and the ODBC Driver Manager don't return warnings when applications run inefficiently. This chapter describes some general guidelines for coding practices that improve ODBC application performance. These guidelines have been compiled by examining the ODBC implementations of numerous shipping ODBC applications. In general, the guidelines described in this chapter improve performance because they accomplish one or more of the following goals:

- Reduce network traffic
- Limit disk I/O
- Optimize application-to-driver interaction
- Simplify queries

If you've read the other coding chapters (Chapters 6 and 7), you'll notice that some of the information here resembles those chapters. While there are some similarities, this chapter focuses on specific information about coding for ODBC.

Managing Connections

Typically, creating a connection is one of the most performance-expensive operations that an application performs. Developers often assume that establishing a connection is a simple request that results in the driver making a single network round trip to the database server to validate a user's credentials. In reality, a connection involves many network round trips between the driver and the database server. For example, when a driver connects to Oracle or Sybase ASE, that connection may require seven to ten network round trips. In addition, the database establishes resources on behalf of the connection, which involves performance-expensive disk I/O and memory allocation.

Your time will be well spent if you sit down and design how to handle connections before implementing them in your application. Use the guidelines in this section to manage your connections more efficiently.

Connecting Efficiently

Database applications use either of the following methods to manage connections:

- Obtain a connection from a connection pool.
- Create a new connection one at a time as needed.

When choosing a method to manage connections, remember the following facts about connections and performance:

- Creating a connection is performance expensive.
- Open connections use a substantial amount of memory on both the database server and the database client.
- Opening numerous connections can contribute to an out-of-memory condition, which causes paging of memory to disk and, thus, overall performance degradation.

Using Connection Pooling

If your application has multiple users and your database server provides sufficient database resources, using connection pooling can provide significant performance gains. Reusing a connection reduces the number of network round trips needed to establish a physical connection between the driver and the database. The performance penalty is paid up front at the time the connection pool is populated with connections. As the connections in the pool are actually used by

the application, performance improves significantly. Obtaining a connection becomes one of the fastest operations an application performs instead of one of the slowest.

Although obtaining connections from a pool is efficient, when your application opens and closes connections impacts the scalability of your application. Open connections just before the user needs them, not sooner, to minimize the time that the user owns the physical connection. Similarly, close connections as soon as the user no longer needs them.

To minimize the number of connections required in a connection pool to service users, you can switch a user associated with a connection to another user if your database driver supports a feature known as reauthentication. Minimizing the number of connections conserves memory and can improve performance. See "Using Reauthentication with Connection Pooling," page 232. See Chapter 8, "Connection Pooling and Statement Pooling," for details about connection pooling.

Establishing Connections One at a Time

Some applications are not good candidates for using connection pooling, particularly if connection reuse is limited. See "When Not to Use Connection Pooling," page 15, for examples.

Performance Tip

If your application does not use connection pooling, avoid connecting and disconnecting multiple times throughout your application to execute SQL statements because of the performance hit your application pays for opening connections. You don't need to open a new connection for each SQL statement your application executes.

Using One Connection for Multiple Statements

When you're using a single connection for multiple statements, your application may have to wait for a connection if it connects to a streaming protocol database. In streaming protocol databases, only one request can be processed at a time over

a single connection; other requests on the same connection must wait for the preceding request to complete. Sybase ASE, Microsoft SQL Server, and MySQL are examples of streaming protocol databases.

In contrast, when connecting to cursor-based protocol databases, the driver tells the database server when to work and how much data to retrieve. Several cursors can use the network, each working in small slices of time. Oracle and DB2 are examples of cursor-based protocol databases. For a more detailed explanation of streaming versus cursor-based protocol databases, see "One Connection for Multiple Statements," page 16.

The advantage of using one connection for multiple statements is that it reduces the overhead of establishing multiple connections, while allowing multiple statements to access the database. The overhead is reduced on both the database server and client machines. The disadvantage is that the application may have to wait to execute a statement until the single connection is available. See "One Connection for Multiple Statements," page 16, for guidelines on using this model of connection management.

Obtaining Database and Driver Information Efficiently

Remember that creating a connection is one of the most performance-expensive operations that an application performs.

Performance Tip

Because of the performance hit your application pays for opening connections, once your application is connected, you should avoid establishing additional connections to gather information about the driver and the database, such as supported data types or database versions, using `SQLGetInfo` and `SQLGetTypeInfo`. For example, some applications establish a connection and then call a routine in a separate DLL or shared library that reconnects and gathers information about the driver and the database.

How often do databases change their supported data types or database version between connections? Because this type of information typically doesn't change between connections and isn't a large amount of information to store, you may want to retrieve and cache the information so the application can access it later.

Managing Transactions

To ensure data integrity, all statements in a transaction are committed or rolled back as a unit. For example, when you use a computer to transfer money from one bank account to another, the request involves a transaction—updating values stored in the database for both accounts. If all parts of that unit of work succeed, the transaction is committed. If any part of that unit of work fails, the transaction is rolled back.

Use the guidelines in this section to help you manage transactions more efficiently.

Managing Commits in Transactions

Committing (and rolling back) transactions is slow because of the disk I/O and, potentially, the number of network round trips required. What does a commit actually involve? The database must write to disk every modification made by a transaction to the database. This is usually a sequential write to a journal file (or log); nevertheless, it involves expensive disk I/O.

In ODBC, the default transaction commit mode is auto-commit. In auto-commit mode, a commit is performed for every SQL statement that requires a request to the database (`Insert`, `Update`, `Delete`, and `Select` statements). When auto-commit mode is used, your application doesn't control when database work is committed. In fact, commits commonly occur when there's actually no real work to commit.

Some databases, such as DB2, don't support auto-commit mode. For these databases, the database driver, by default, sends a commit request to the database after every successful operation (SQL statement). The commit request equates to a network round trip between the driver and the database. The round trip to the database occurs even though the application didn't request the commit and even if the operation made no changes to the database. For example, the driver makes a network round trip even when a `Select` statement is executed.

Let's look at the following ODBC code, which doesn't turn off auto-commit mode. Comments in the code show when commits occur if the driver or the database performs commits automatically:

```
/* For conciseness, this code omits error checking */

/* Allocate a statement handle */
rc = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
```

```
/* Prepare an INSERT statement for multiple executions */
strcpy (sqlStatement, "INSERT INTO employees " +
   "VALUES (?, ?, ?)");
rc = SQLPrepare((SQLHSTMT)hstmt, sqlStatement, SQL_NTS);

/* Bind parameters */
rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT,
                    SQL_C_SLONG, SQL_INTEGER, 10, 0,
                    &id, sizeof(id), NULL);
rc = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT,
                    SQL_C_CHAR, SQL_CHAR, 20, 0,
                    name, sizeof(name), NULL);
rc = SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT,
                    SQL_C_SLONG, SQL_INTEGER, 10, 0,
                    &salary, sizeof(salary), NULL);

/* Set parameter values before execution */
id = 20;
strcpy(name, "Employee20");
salary = 100000;
rc = SQLExecute(hstmt);

/* A commit occurs because auto-commit is on */

/* Change parameter values for the next execution */
id = 21;
strcpy(name, "Employee21");
salary = 150000;
rc = SQLExecute(hstmt);

/* A commit occurs because auto-commit is on */

/* Reset parameter bindings */
rc = SQLFreeStmt((SQLHSTMT)hstmt, SQL_RESET_PARAMS);
strcpy(sqlStatement, “SELECT id, name, salary “ +
   “FROM employees”);
```

```
/* Execute a SELECT statement. A prepare is unnecessary
   because it's executed only once. */
rc = SQLExecDirect((SQLHSTMT)hstmt, sqlStatement, SQL_NTS);

/* Fetch the first row */
rc = SQLFetch(hstmt);
while (rc != SQL_NO_DATA_FOUND) {

/* All rows are returned when fetch
   returns SQL_NO_DATA_FOUND */

   /* Get the data for each column in the result set row */
   rc = SQLGetData (hstmt, 1, SQL_INTEGER, &id,
                    sizeof(id), NULL);
   rc = SQLGetData (hstmt, 2, SQL_VARCHAR, &name,
                    sizeof(name), NULL);
   rc = SQLGetData (hstmt, 3, SQL_INTEGER, &salary,
                    sizeof(salary), NULL);
   printf("\nID: %d Name: %s Salary: %d", id, name, salary);

   /* Fetch the next row of data */
   rc = SQLFetch(hstmt);
   }

/* Close the cursor */
rc = SQLFreeStmt ((SQLHSTMT)hstmt, SQL_CLOSE);

/* Whether a commit occurs after a SELECT statement
   because auto-commit is on depends on the driver.
   It's safest to assume a commit occurs here.     */

/* Prepare the UPDATE statement for multiple executions */
strcpy (sqlStatement,
        "UPDATE employees SET salary = salary * 1.05" +
        "WHERE id = ?");
```

```
rc = SQLPrepare ((SQLHSTMT)hstmt, sqlStatement, SQL_NTS);

/* Bind parameter */
rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT,
                      SQL_C_LONG, SQL_INTEGER, 10, 0,
                      &index, sizeof(index), NULL);

for (index = 0; index < 10; index++) {

   /* Execute the UPDATE statement for each
      value of index between 0 and 9 */
   rc = SQLExecute (hstmt);

/* Because auto-commit is on, a commit occurs each time
   through loop for a total of 10 commits */
   }

/* Reset parameter bindings */
rc = SQLFreeStmt ((SQLHSTMT)hstmt, SQL_RESET_PARAMS);

/* Execute a SELECT statement. A prepare is unnecessary
   because it's only executed once. */
strcpy(sqlStatement, "SELECT id, name, salary" +
  "FROM employees");
rc = SQLExecDirect ((SQLHSTMT)hstmt, sqlStatement, SQL_NTS);

/* Fetch the first row */
rc = SQLFetch(hstmt);
while (rc != SQL_NO_DATA_FOUND) {

/* All rows are returned when fetch
   returns SQL_NO_DATA_FOUND */
```

```
  /* Get the data for each column in the result set row */
  rc = SQLGetData (hstmt, 1, SQL_INTEGER, &id,
                   sizeof(id), NULL);
  rc = SQLGetData (hstmt, 2, SQL_VARCHAR, &name,
                   sizeof(name), NULL);
  rc = SQLGetData (hstmt,3,SQL_INTEGER,&salary,
                   sizeof(salary), NULL);
   printf("\nID: %d Name: %s Salary: %d", id, name, salary);

   /* Fetch the next row of data */
   rc = SQLFetch(hstmt);
   }

/* Close the cursor */
rc = SQLFreeStmt ((SQLHSTMT)hstmt, SQL_CLOSE);

/* Whether a commit occurs after a SELECT statement
   because auto-commit is on depends on the driver.
   It's safest to assume a commit occurs here.     */
```

Performance Tip

Because of the significant amount of disk I/O on the database server required to commit every operation and the extra network round trips that occur between the driver and the database server, it's a good idea to turn off auto-commit mode in your application and use manual commits instead. Using manual commits allows your application to control when database work is committed, which provides dramatically better performance. To turn off auto-commit mode, use the `SQLSetConnectAttr` function, for example, `SQLSetConnectAttr(hstmt, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF)`.

For example, let's look at the following ODBC code. It's identical to the previous ODBC code except that it turns off auto-commit mode and uses manual commits:

```
/* For conciseness, this code omits error checking */

/* Allocate a statement handle */
rc = SQLAllocStmt((SQLHDBC)hdbc, (SQLHSTMT *)&hstmt);

/* Turn auto-commit off */
rc = SQLSetConnectAttr (hdbc, SQL_AUTOCOMMIT,
                        SQL_AUTOCOMMIT_OFF);

/* Prepare an INSERT statement for multiple executions */
strcpy (sqlStatement, "INSERT INTO employees" +
  "VALUES (?, ?, ?)");
rc = SQLPrepare((SQLHSTMT)hstmt, sqlStatement, SQL_NTS);

/* Bind parameters */
rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT,
                      SQL_C_SLONG, SQL_INTEGER, 10, 0,
                      &id, sizeof(id), NULL);
rc = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT,
                      SQL_C_CHAR, SQL_CHAR, 20, 0,
                      name, sizeof(name), NULL);
rc = SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT,
                      SQL_C_SLONG, SQL_INTEGER, 10, 0,
                      &salary, sizeof(salary), NULL);

/* Set parameter values before execution */
id = 20;
strcpy(name,"Employee20");
salary = 100000;
rc = SQLExecute(hstmt);

/* Change parameter values for the next execution */
id = 21;
strcpy(name,"Employee21");
```

```
salary = 150000;
rc = SQLExecute(hstmt);

/* Reset parameter bindings */
rc = SQLFreeStmt(hstmt, SQL_RESET_PARAMS);

/* Manual commit */
rc = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);

/* Execute a SELECT statement. A prepare is unnecessary
   because it's only executed once. */
strcpy(sqlStatement, "SELECT id, name, salary" +
  "FROM employees");
rc = SQLExecDirect((SQLHSTMT)hstmt, sqlStatement, SQL_NTS);

/* Fetch the first row */
rc = SQLFetch(hstmt);
while (rc != SQL_NO_DATA_FOUND) {

/* All rows are returned when fetch
  returns SQL_NO_DATA_FOUND */

   /* Get the data for each column in the result set row */
   rc = SQLGetData (hstmt, 1, SQL_INTEGER, &id,
                    sizeof(id), NULL);
   rc = SQLGetData (hstmt, 2, SQL_VARCHAR, &name,
                    sizeof(name), NULL);
   rc = SQLGetData (hstmt, 3, SQL_INTEGER, &salary,
                    sizeof(salary), NULL);
   printf("\nID: %d Name: %s Salary: %d", id, name, salary);

   /* Fetch the next row of data */
   rc = SQLFetch(hstmt);
   }
```

```
/* Close the cursor */
rc = SQLFreeStmt ((SQLHSTMT)hstmt, SQL_CLOSE);

strcpy (sqlStatement,
        "UPDATE employees SET salary = salary * 1.05" +
        "WHERE id = ?");

/* Prepare the UPDATE statement for multiple executions */
rc = SQLPrepare ((SQLHSTMT)hstmt, sqlStatement, SQL_NTS);

/* Bind parameter */
rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT,
                      SQL_C_SLONG, SQL_INTEGER, 10, 0,
                      &index, sizeof(index), NULL);

for (index = 0; index < 10; index++) {

   /* Execute the UPDATE statement for each
      value of index between 0 and 9 */
   rc = SQLExecute (hstmt);
   }

/* Manual commit */
rc = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);

/* Reset parameter bindings */
rc = SQLFreeStmt ((SQLHSTMT)hstmt, SQL_RESET_PARAMS);

/* Execute a SELECT statement. A prepare is unnecessary
   because it's only executed once. */
strcpy(sqlStatement, "SELECT id, name, salary" +
  "FROM employees");
rc = SQLExecDirect ((SQLHSTMT)hstmt, sqlStatement, SQL_NTS);
```

```
/* Fetch the first row */
rc = SQLFetch(hstmt);
while (rc != SQL_NO_DATA_FOUND) {

/* All rows are returned when fetch
   returns SQL_NO_DATA_FOUND */

   /* Get the data for each column in the result set row */
   rc = SQLGetData (hstmt, 1, SQL_INTEGER, &id,
                    sizeof(id), NULL);
   rc = SQLGetData (hstmt, 2, SQL_VARCHAR, &name,
                    sizeof(name), NULL);
   rc = SQLGetData (hstmt,3,SQL_INTEGER,&salary,
                    sizeof(salary), NULL);
   printf("\nID: %d Name: %s Salary: %d", id, name, salary);

   /* Fetch the next row of data */
   rc = SQLFetch(hstmt);
   }

/* Close the cursor */
rc = SQLFreeStmt ((SQLHSTMT)hstmt, SQL_CLOSE);

/* Manual commit */
rc = SQLEndTran (SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
```

See "Managing Commits in Transactions," page 22, for information on when to commit work if you've turned off auto-commit mode.

Choosing the Right Transaction Model

Which type of transaction should you use: local or distributed? A local transaction accesses and updates data on a single database. A distributed transaction accesses and updates data on multiple databases; therefore, it must be coordinated among those databases.

Performance Tip

Distributed transactions, as defined by ODBC and the Microsoft Distributed Transaction Coordinator (DTC), are substantially slower than local transactions because of the logging and network round trips needed to communicate between all the components involved in the distributed transaction. Unless distributed transactions are required, you should use local transactions.

Be aware that the default transaction behavior of many COM+ components uses distributed transactions, so changing that default transaction behavior to local transactions as shown can improve performance.

```
// Disable MTS Transactions.
XACTOPT options[1] = {XACTSTAT_NONE,"NOT SUPPORTED"};
hr = Itxoptions->SetOptions(options);
```

See "Transaction Management," page 21, for more information about performance and transactions.

Executing SQL Statements

Use the guidelines in this section to help you select which ODBC functions will give you the best performance when executing SQL statements.

Using Stored Procedures

Database drivers can call stored procedures on the database using either of the following methods:

- Execute the procedure the same way as any other SQL statement. The database parses the SQL statement, validates argument types, and converts arguments into the correct data types.
- Invoke a Remote Procedure Call (RPC) directly in the database. The database skips the parsing and optimization that executing a SQL statement requires.

Performance Tip

Call stored procedures by invoking an RPC with parameter markers for arguments instead of using literal arguments. Because the database skips the parsing and optimization required in executing the stored procedure as a SQL statement, performance is significantly improved.

Remember that SQL is always sent to the database as a character string. For example, consider the following stored procedure call, which passes a literal argument to the stored procedure:

```
{call getCustName (12345)}
```

Although the argument to `getCustName()` is an integer, the argument is passed inside a character string to the database, namely `{call getCustName (12345)}`. The database parses the SQL statement, isolates the single argument value of 12345, and converts the string `12345` into an integer value before executing the procedure as a SQL language event. Using an RPC on the database, your application can pass the parameters to the RPC. The driver sends a database protocol packet that contains the parameters in their native data type formats, skipping the parsing and optimization required to execute the stored procedure as a SQL statement. Compare the following examples.

Example A: Not Using a Server-Side RPC

The stored procedure `getCustName` is not optimized to use a server-side RPC. The database treats the SQL stored procedure execution request as a normal SQL language event, which includes parsing the statement, validating argument types, and converting arguments into the correct data types before executing the procedure.

```
strcpy (sqlStatement,"{call getCustName (12345)}");
rc = SQLPrepare((SQLHSTMT)hstmt, sqlStatement, SQL_NTS);
rc = SQLExecute(hstmt);
```

Example B: Using a Server-Side RPC

The stored procedure `getCustName` is optimized to use a server-side RPC. Because the application avoids literal arguments and calls the procedure by specifying arguments as parameters, the driver optimizes the execution by invoking the stored procedure directly on the database as an RPC. The SQL language processing by the database is avoided, and execution time is faster.

```
strcpy (sqlStatement,"{call getCustName (?)}");
rc = SQLPrepare((SQLHSTMT)hstmt, sqlStatement, SQL_NTS);
rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT,
                      SQL_C_LONG, SQL_INTEGER, 10, 0,
                      &id, sizeof(id), NULL);
id = 12345;
rc = SQLExecute(hstmt);
```

Why doesn't the driver parse and automatically change the SQL stored procedure call when it encounters a literal argument so that it can execute the stored procedure using an RPC? Consider this example:

```
{call getCustname (12345)}
```

The driver doesn't know if the value `12345` represents an integer, a decimal, a smallint, a bigint, or another numeric data type. To determine the correct data type for packaging the RPC request, the driver must make an expensive network round trip to the database server. The overhead needed to determine the true data type of the literal argument far outweighs the benefit of trying to execute the request as an RPC.

Using Statements Versus Prepared Statements

Most applications have a certain set of SQL statements that are executed multiple times and a few SQL statements that are executed only once or twice during the life of the application. Choose the `SQLExecDirect` function or the `SQLPrepare`/`SQLExecute` functions depending on how frequently you plan to execute the SQL statement.

The `SQLExecDirect` function is optimized for a SQL statement that is only executed once. In contrast, the `SQLPrepare`/`SQLExecute` functions are optimized for SQL statements that use parameter markers and are executed multiple times. Although the overhead for the initial execution of a prepared statement is high, the advantage is realized with subsequent executions of the SQL statement.

Using the `SQLPrepare`/`SQLExecute` functions typically results in at least two network round trips to the database server:

- One network round trip to parse and optimize the statement
- One or more network round trips to execute the statement and retrieve results

Performance Tip

If your application makes a request only once during its life span, using the SQLExecDirect function is a better choice than using the SQLPrepare/SQLExecute function because SQLExecDirect results in only a single network round trip. Remember, reducing network communication typically provides the most performance gain. For example, if you have an application that runs an end-of-day sales report, send the query that generates the data for that report to the database using the `SQLExecDirect` function, not the `SQLPrepare`/`SQLExecute` function.

See "SQL Statements," page 27, for more information about statements versus prepared statements.

Using Arrays of Parameters

Updating large amounts of data typically is done by preparing an `Insert` statement and executing that statement multiple times, resulting in many network round trips.

Performance Tip

To reduce the number of network round trips when updating large amounts of data, you can send multiple `Insert` statements to the database at a time using the `SQLSetStmtAttr` function with the following arguments: `SQL_ATTR_PARAMSET_SIZE` sets the array size of the parameter, `SQL_ATTR_PARAMS_PROCESSED_PTR` assigns a variable filled by `SQLExecute` (containing the number of rows that are inserted), and `SQL_ATTR_PARAM_STATUS_PTR` points to an array in which status information for each row of parameter values is retrieved.

With ODBC 3.*x*, calls to `SQLSetStmtAttr` with the `SQL_ATTR_PARAMSET_SIZE`, `SQL_ATTR_PARAMS_PROCESSED_PTR`, and `SQL_ATTR_PARAM_STATUS_PTR` arguments supersede 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 round trip. For example, let's compare the following examples.

Example A: Executing a Prepared Statement Multiple Times

A prepared statement is used to execute an `Insert` statement multiple times, requiring 101 network round trips to perform 100 `Insert` operations: 1 round trip to prepare the statement and 100 additional round trips to execute its iterations.

```
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);
```

Example B: Arrays of Parameters

When arrays of parameters are used to consolidate 100 `Insert` operations, only two network round trips are required: one to prepare the statement and another to execute the array. Although arrays of parameters use more CPU cycles, performance is gained by reducing the number of network round trips.

```
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_PTR,
   &rows_processed, SQL_IS_POINTER);
// Specify an array in which to retrieve 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);
```

Using the Cursor Library

The ODBC cursor library is a component of Microsoft Data Access Components (MDAC) and is used to implement static cursors (one type of scrollable cursor) for drivers that normally don't support them.

Performance Tip

If your ODBC driver supports scrollable cursors, don't code your application to load the ODBC cursor library. Although the cursor library provides support for static cursors, the cursor library also creates temporary log files on the user's local disk drive. Because of the disk I/O required to create these temporary log files, using the ODBC cursor library slows performance.

What if you don't know whether your driver supports scrollable cursors? Using the following code ensures that the ODBC cursor library is only used when the driver doesn't support scrollable cursors:

```
rc = SQLSetConnectAttr (hstmt, SQL_ATTR_ODBC_CURSORS,
    SQL_CUR_USE_IF_NEEDED, SQL_IS_INTEGER);
```

Retrieving Data

Retrieve only the data you need, and choose the most efficient method to retrieve that data. Use the guidelines in this section to optimize your performance when retrieving data.

Retrieving Long Data

Retrieving long data—such as large XML data, long varchar/text, long varbinary, Clobs, and Blobs—across a network is slow and resource intensive. Most users really don't want to see long data. For example, consider the user interface of an employee directory application that allows the user to look up an employee's phone extension and department, and optionally, view an employee's photograph by clicking the name of the employee.

Employee	Phone	Dept
Harding	X4568	Manager
Hoover	X4324	Sales
Taft	X4569	Sales
Lincoln	X4329	Tech

Retrieving each employee's photograph would slow performance unnecessarily. If the user does want to see the photograph, he can click the employee name and the application can query the database again, specifying only the long columns in the `Select` list. This method allows users to retrieve result sets without paying a high performance penalty for network traffic.

Although excluding long data from the `Select` list is the best approach, some applications do not formulate the `Select` list before sending the query to the driver (that is, some applications use `SELECT * FROM table ...`). If the `Select` list contains long data, the driver is forced to retrieve that long data, even if the application never requests the long data from the result set. When possible, use a method that does not retrieve all columns of the table. For example, consider the following code:

```
rc = SQLExecDirect (hstmt, "SELECT * FROM employees" +
                   "WHERE SSID = '999-99-2222'", SQL_NTS);
rc = SQLFetch (hstmt);
```

When a query is executed, the driver has no way to determine which result columns the application will use; an application may fetch any result column that is retrieved. When the driver processes a `SQLFetch` or `SQLExtendedFetch` request, it retrieves at least one, and often multiple, result rows from the database across the network. A result row contains all the column values for each row. What if one of the columns includes long data such as an employee photograph? Performance would slow considerably.

Performance Tip

Because retrieving long data across the network negatively affects performance, design your application to exclude long data from the Select list.

Limiting the `Select` list to contain only the name column results in a faster performing query at runtime. For example:

```
rc = SQLExecDirect (hstmt, "SELECT name FROM employees" +
                   "WHERE SSID = '999-99-2222'", SQL_NTS);
rc = SQLFetch(hstmt);
rc = SQLGetData(hstmt, 1, ...);
```

Limiting the Amount of Data Retrieved

If your application executes a query that retrieves five rows when it needs only two, application performance suffers, especially if the unnecessary rows include long data.

Performance Tip

One of the easiest ways to improve performance is to limit the amount of network traffic between the driver and the database server—optimally by writing SQL queries that instruct the driver to retrieve from the database only the data that the application requires.

Make sure that your `Select` statements use a `Where` clause to limit the amount of data that is retrieved. Even when using a `Where` clause, a `Select` statement that does not adequately restrict its request could retrieve hundreds of rows of data. For example, if you want data from the employees table for each manager hired in recent years, your application could execute the following statement, and subsequently, filter out the rows of employees who are not managers:

```
SELECT * FROM employees
WHERE hiredate > 2000
```

However, suppose the employees table contains a column that stores photographs of each employee. In this case, retrieving extra rows is extremely expensive to your application performance. Let the database filter the request for you and avoid sending extra data that you don't need across the network. The following query uses a better approach, limiting the data retrieved and improving performance:

```
SELECT * FROM employees
WHERE hiredate > 2003 AND job_title='Manager'
```

Sometimes applications need to use SQL queries that generate a large amount of network traffic. For example, consider an application that displays information from support case histories, which each contains a 10MB log file. Does the user really need to see the entire contents of the log file? If not, performance would improve if the application displayed only the first 1MB of the log file.

Performance Tip

When you cannot avoid retrieving data that generates a large amount of network traffic, your application can still control the amount of data being sent from the database to the driver by limiting the number of rows sent across the network and reducing the size of each row sent across the network.

Suppose you have a GUI-based application, and each screen can display no more than 20 rows of data. It's easy to construct a query that may retrieve a million rows, such as `SELECT * FROM employees`, but it's hard to imagine a scenario where a query that retrieves a million rows would be useful. When designing applications, it's good practice to call the `SQLSetStmtAttr` function with the `SQL_ATTR_MAX_ROWS` option as a fail-safe to limit the number of rows that a query can retrieve. For example, if an application calls `SQLSetStmt(SQL_ATTR_MAX_ROWS, 10000, 0)`, no query will retrieve more than 10,000 rows.

In addition, calling the `SQLSetStmtAttr` function with the `SQL_ATTR_MAX_LENGTH` option limits the bytes of data that can be retrieved for a column value with the following data types:

- Binary
- Varbinary
- Longvarbinary
- Char
- Varchar
- Longvarchar

For example, consider an application that allows users to select from a repository of technical articles. Rather than retrieve and display the entire article, the application can call `SQLSetStmtAttr(SQL_ATTR_MAX_LENGTH, 153600, 0)` to retrieve only the first 150KB of text to the application—enough to give users a reasonable preview of the article.

Using Bound Columns

Data can be retrieved from the database using either the `SQLBindCol` function or the `SQLGetData` function. When SQLBindCol is called, it associates, or binds, a variable to a column in the result set. Nothing is sent to the database.

SQLBindCol tells the driver to remember the addresses of the variables, which the driver will use to store the data when it is actually retrieved. When SQLFetch is executed, the driver places the data into the addresses of the variables specified by SQLBindCol. In contrast, SQLGetData returns data directly into variables. It's commonly called to retrieve long data, which often exceeds the length of a single buffer and must be retrieved in parts.

Performance Tip

Retrieving data using the SQLBindCol function instead of using the SQLGetData function reduces the number of ODBC calls, and ultimately the number of network round trips, improving performance.

The following code uses the SQLGetData function to retrieve data:

```
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));
```

If the query retrieves 90 result rows, 1,891 ODBC calls are made (20 calls to SQLGetData × 90 result rows + 91 calls to SQLFetch).

The following code uses the SQLBindCol function 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 is reduced from 1,891 to 111 (20 calls to SQLBindCol + 91 calls to SQLFetch). In addition to reducing the number of calls required, many drivers optimize how SQLBindCol is used by binding result information directly from the database 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 requests that the information from the database be placed directly into the user's buffer.

Using SQLExtendedFetch Instead of SQLFetch

Most ODBC drivers now support SQLExtendedFetch for forward-only cursors. Yet, most ODBC applications continue to use SQLFetch to fetch data.

Performance Tip

Using the SQLExtendedFetch function instead of SQLFetch to fetch data reduces the number of ODBC calls, and ultimately the number of network round trips, and simplifies coding. Using SQLExtendedFetch results in better performance and more maintainable code.

Again, consider the same example we used in the section, "Using Bound Columns," page 145, but 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));
```

The number of ODBC calls made by the application has been reduced from 1,891 to 4 (1 SQLSetStmtOption + 1 SQLExecDirect + 1 SQLBindCol + 1 SQLExtendedFetch). Besides reducing the ODBC call load, some ODBC drivers

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

For ODBC drivers that do not support `SQLExtendedFetch`, your application can enable forward-only cursors using the ODBC cursor library by calling `SQLSetConnectAttr`. Using the cursor library won't improve performance, but it also won't decrease application response time when using forward-only cursors because no logging is required. For scrollable cursors, it's a different story (see "Using the Cursor Library," page 141). In addition, using the cursor library when `SQLExtendedFetch` is not supported natively by the driver simplifies code because the application can always depend on `SQLExtendedFetch` being available. The application doesn't require two algorithms (one using `SQLExtendedFetch` and one using `SQLFetch`).

Determining the Number of Rows in a Result Set

ODBC defines two types of cursors:

- Forward-only
- Scrollable (static, keyset-driven, dynamic, and mixed)

Scrollable cursors let you go both forward and backward through a result set. However, because of limited support for server-side scrollable cursors in many database systems, ODBC drivers often emulate scrollable cursors, storing rows from a scrollable result set in a cache on the machine where the driver resides (client or application server).

Unless you are certain that the database natively supports using a scrollable result set, do not call the `SQLExtendedFetch` function to find out how many rows the result set contains. For drivers that emulate scrollable cursors, calling `SQLExtendedFetch` causes the driver to retrieve all results across the network to reach the last row. This emulated model of scrollable cursors provides flexibility for the developer but comes with a performance penalty until the client cache of rows is fully populated. Instead of calling `SQLExtendedFetch` to determine the number of rows, count the rows by iterating through the result set or obtain the number of rows by submitting a `Select` statement with the `Count` function. For example:

```
SELECT COUNT(*) FROM employees
```

Unfortunately, there's no easy way to tell if a database driver uses native server-side scrollable cursors or emulates this functionality. For Oracle or

MySQL, you know the driver emulates scrollable cursors, but for other databases, it's more complicated. See "Using Scrollable Cursors," page 36, for details about which common databases support server-side scrollable cursors and how database drivers emulate scrollable cursors.

Performance Tip

In general, do not write code that relies on the number of result rows from a query because drivers often must retrieve all rows in a result set to determine how many rows the query will return.

Choosing the Right Data Type

When designing your database schema, it's obvious that you need to think about the impact of storage requirements on the database server. Less obvious, but just as important, you need to think about the network traffic required to move data in its native format to and from the ODBC driver. Retrieving and sending certain data types across the network can increase or decrease network traffic.

Performance Tip

For multiuser, multivolume applications, billions, or even trillions, of network packets can move between the driver and the database server over the course of a day. Choosing data types that are processed efficiently can incrementally provide a measurable gain in performance.

See "Choosing the Right Data Type," page 34, for information about which data types are processed faster than others.

Updating Data

Use the guidelines in this section to manage your updates more efficiently.

Using `SQLSpecialColumns` to Optimize Updates and Deletes

Many databases have hidden columns, named **pseudo-columns,** that represent a unique key associated with every row in a table. Typically, pseudo-columns in a

SQL statement provide the fastest way to access a row because they usually point to the exact location of the physical record.

Performance Tip

Use `SQLSpecialColumns` to identify the most optimal columns, typically pseudo-columns, to use in the `Where` clause for updating data.

Some applications, such as an application that forms a `Where` clause consisting of a subset of the column values retrieved in the result set, cannot be designed to take advantage of positioned updates and deletes. Some applications may formulate the `Where` clause by using searchable result columns or by calling `SQLStatistics` to find columns that may be part of a unique index. These methods usually work but can result in fairly complex queries. For example:

```
rc = SQLExecDirect (hstmt, "SELECT first_name, last_name," +
                   "ssn, address, city, state, zip" +
                   "FROM employees", SQL_NTS);
// fetch data using complex query
...
rc = SQLExecDirect (hstmt, "UPDATE employees SET address = ?" +
                   "WHERE first_name = ? AND last_name = ?" +
                   "AND ssn = ? AND address = ? AND city = ? AND" +
                   "state = ? AND zip = ?", SQL_NTS);
```

Many databases support pseudo-columns that are not explicitly defined in the table definition but are hidden columns of every table (for example, `ROWID` for Oracle). Because pseudo-columns are not part of the explicit table definition, they're not retrieved when `SQLColumns` is called. To determine if pseudo-columns exist, your application must call `SQLSpecialColumns`. For example:

```
...
rc = SQLSpecialColumns (hstmt, SQL_BEST_ROWID, ...);
...
```

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

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

Using Catalog Functions

Catalog functions retrieve information about a result set, such as the number and type of columns. Because catalog functions are slow compared to other ODBC functions, using them frequently can impair performance. Use the guidelines in this section to optimize performance when selecting and using catalog functions.

Minimizing the Use of Catalog Functions

Compared to other ODBC functions, catalog functions that generate result sets are slow. To retrieve all result column information mandated by the ODBC specification, an ODBC driver often must perform multiple or complex queries to retrieve the result set for a single call to a catalog function.

Performance Tip

Although it's almost impossible to write an ODBC application without using a catalog function, you can improve performance by minimizing their use.

In addition to avoid executing catalog functions multiple times, you should cache information retrieved from result sets generated by catalog functions. For example, call `SQLGetTypeInfo` once, and cache the elements of the result set that your application depends on. It's unlikely that any application will use all elements of the result set generated by a catalog function, so the cache of information shouldn't be difficult to maintain.

Avoiding Search Patterns

Catalog functions support arguments that can limit the amount of data retrieved. Using null values or search patterns, such as `%A%`, for these arguments often generates time-consuming queries. In addition, network traffic can increase because of unnecessary results.

Performance Tip

Always supply as many non-null arguments as possible to result sets that generate catalog functions.

In the following example, an application uses the `SQLTables` function to determine whether the table named `WSTable` exists and provides null values for most of the arguments:

```
rc = SQLTables(hstmt, null, 0, null, 0, "WSTable",
               SQL_NTS, null, 0);
```

The driver interprets the request as follows: Retrieve all tables, views, system tables, synonyms, temporary tables, and aliases named `WSTable` that exist in any database schema in the database catalog.

In contrast, the following request provides non-null values for all arguments, allowing the driver to process the request more efficiently:

```
rc = SQLTables(hstmt, "cat1", SQL_NTS, "johng", SQL_NTS,
               "WSTable", SQL_NTS, "Table", SQL_NTS);
```

The driver interprets the request as follows: Retrieve all tables in catalog `"cat1"` that are named `"WSTable"` and owned by `"johng."` No synonyms, views, system tables, aliases, or temporary tables are retrieved.

Sometimes little is known about the object that you are requesting information for. Any information that the application can provide the driver when calling catalog functions can result in improved performance and reliability.

Using a Dummy Query to Determine Table Characteristics

Sometimes you need information about columns in the database table, such as column names, column data types, and column precision and scale. For example, an application that allows users to choose which columns to select may need to request the names of each column in the database table.

Performance Tip

To determine characteristics about a database table, avoid using the `SQLColumns` function. Instead, use a dummy query inside a prepared statement that executes the `SQLDescribeCol` function. Only use the `SQLColumns` function when you cannot obtain the requested information from result set metadata (for example, using the table column default values).

The following examples show the benefit of using the `SQLDescribeCol` function over the `SQLColumns` function.

Example A: `SQLColumns` Function

A potentially complex query is prepared and executed, the result description information is formulated, the driver retrieves the result rows, and the application fetches the result. This method results in increased CPU use and network communication.

```
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
```

Example B: `SQLDescribeCol` Function

A simple query that retrieves result set information is prepared, but the query is not executed and result rows are not retrieved by the driver. Only information about the result set is retrieved (the same information retrieved by `SQLColumns` in Example A).

```
// 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.
```

What if the database server, such as a Microsoft SQL Server server does not support prepared statements by default? The performance of Example A wouldn't change, but the performance of Example B would decrease slightly because the dummy query is evaluated in addition to being prepared. Because the `Where` clause of the query always evaluates to `FALSE`, the query generates no result rows and executes the statement without retrieving result rows. So, even with a slight decrease in performance, Example B still outperforms Example A.

Summary

The performance of ODBC applications can suffer if they fail to reduce network traffic, limit disk I/O, simplify queries, and optimize the interaction between the application and driver. Reducing network communication probably is the most important technique for improving performance. For example, when you need to update large amounts of data, using arrays of parameters rather than executing an `Insert` statement multiple times reduces the number of network round trips required to complete the operation.

Typically, creating a connection is the most performance-expensive task your application performs. Connection pooling can help you manage your connections efficiently, particularly if your application has numerous users. Regardless of whether your application uses connection pooling, make sure that your application closes connections immediately after the user is finished with them.

Making smart choices about how to handle transactions can also improve performance. For example, using manual commits instead of auto-commit mode provides better control over when work is committed. Similarly, if you don't need the protection of distributed transactions, using local transactions can improve performance.

Inefficient SQL queries slow the performance of ODBC applications. Some SQL queries don't filter data, causing the driver to retrieve unnecessary data. Your application pays a huge penalty in performance when that unnecessary data is long data, such as data stored as a Blob or Clob. Even well-formed SQL queries can be more or less effective depending on how they are executed. For example, using `SQLExtendedFetch` instead of `SQLFetch` and using `SQLBindCol` instead of `SQLGetData` reduces ODBC calls and improves performance.